--

[string 1](#_Toc14857)

[综述 1](#_Toc16144)

[js正则表达式之exec()、test()、match()、search()方法 2](#_Toc17547)

[test 2](#_Toc30676)

[match 2](#_Toc20088)

[search 2](#_Toc17980)

[exec 2](#_Toc17793)

[split 方法 3](#_Toc27255)

[replace 方法 [不会改变原字符] 3](#_Toc11061)

[简介 3](#_Toc21065)

[$符号 3](#_Toc30864)

[demo 4](#_Toc27874)

['border-bottom-color' --> borderBottomColor  4](#_Toc28004)

[添加千位符 4](#_Toc23206)

[正则 5](#_Toc28921)

[array 5](#_Toc21608)

[API 5](#_Toc13527)

[属性 6](#_Toc14727)

[es5 6](#_Toc9968)

[简述 6](#_Toc25591)

[forEach 6](#_Toc30223)

[map  7](#_Toc6515)

[filter  7](#_Toc17071)

[some 7](#_Toc27507)

[every  7](#_Toc10157)

[indexOf 7](#_Toc9114)

[lastIndexOf 7](#_Toc24949)

[reduce 8](#_Toc2222)

[reduceRight 8](#_Toc20371)

[es6 8](#_Toc26728)

[简述 8](#_Toc11215)

[from 8](#_Toc29845)

[of 9](#_Toc27462)

[fill 9](#_Toc13914)

[find 9](#_Toc8470)

[findIndex 9](#_Toc12455)

[includes 9](#_Toc14230)

[keys values entries 9](#_Toc21265)

[copyWithin 9](#_Toc15417)

[解构 10](#_Toc10412)

[合并数组方法concat push对比 10](#_Toc7042)

[concat 10](#_Toc19109)

[push 10](#_Toc21948)

[splice 10](#_Toc7331)

[在类数组对象上重用数组方法: 10](#_Toc14934)

[其他 10](#_Toc18747)

[数组排序 11](#_Toc26369)

[排序1 11](#_Toc26364)

[练习 11](#_Toc28950)

[Array.indexOf() 11](#_Toc13018)

[判断一个对象是否为一个数组 11](#_Toc4807)

[for…in 遍历数组的隐藏问题 11](#_Toc29447)

[JS数组元素删除:delete splice 11](#_Toc17183)

[判断一个对象是否为数组 11](#_Toc28026)

[数组中随机获取成员 12](#_Toc5872)

[jquery工具函数-each map... 12](#_Toc30637)

[each 12](#_Toc31244)

[map 12](#_Toc4446)

[grep 12](#_Toc2951)

[extend 12](#_Toc19038)

[merge 12](#_Toc31959)

[unique 12](#_Toc29604)

[判断 12](#_Toc4865)

[其他 13](#_Toc24138)

[Math 13](#_Toc8768)

[Date 13](#_Toc16155)

[new Date() 13](#_Toc24515)

[Date.parse() 14](#_Toc17364)

[tostring 14](#_Toc627)

[getYear 14](#_Toc3924)

[getFullYear 14](#_Toc13029)

[其他 14](#_Toc15427)

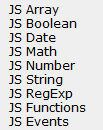
[一行代码 14](#_Toc3045)

[常用的jsLib 14](#_Toc16892)

[一行代码分析 14](#_Toc14390)

[谷歌调试 15](#_Toc13755)

[json 15](#_Toc24447)



虽有嘉肴,弗食,不知其旨也;

虽有至道,弗学,不知其善也。

是故学然后知不足,教然后知困。

知不足,然后能自反也;

知困,然后能自强也。

<http://www.css88.com/archives/6236>

JavaScript中undefined和null的区别

<http://www.zcfy.cc/article/67-useful-tools-libraries-and-resources-for-saving-your-time-as-a-web-developer-4174.html?t=new>

67 个拯救前端开发者的工具、库和资源

JavaScript从基础到弃坑 - 数据类型转换

<http://iwangfan.site/blog/data-type-conversion/>

## string

综述

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 获取字符串的长度 | [js]  str.length;  Array.length | | | [java]  String.length();  Array.length; | | | [Jquery]  获取数组的长度：  arr.size()  arr.length | |
| 显示的方法 | log("name".big()); log("name".small()); log("name".bold()); log("name".italics()); log("name".blink()); log("name".fixed()); log("name".strike()); log("name".fontcolor('red')); log("name".fontsize(16)); log("name".toLowerCase()); log("name".toUpperCase()); log("name".sub()); log("name".sup()); log("name".link("../")); | | | | <big>name</big>  <small>name</small>  <b>name</b>  <i>name</i>  <blink>name</blink>  <tt>name</tt>  <strike>name</strike>  <font color="red">name</font>  <font size="16">name</font>  name  NAME  <sub>name</sub>  <sup>name</sup>  <a href="../">name</a> | | | |
|  | | | |  | | | |
| 支持正则 | match()  search()  exec()  test()  split()  replace() | match() 方法可在字符串内检索指定的值，或找到一个或多个正则表达式的匹配；如果没有找到则返回null；  search() 方法用于检索字符串中指定的子字符串，或检索与正则表达式相匹配的子字符串；返回第一个与 regexp 相匹配的子串的起始位置；如果没有找到任何匹配的子串，则返回 -1；  search（）获取的是相应的下标，match（）获取的是数组（值）； | | | | | | |
| split() 方法用于把一个字符串分割成字符串数组。  String.split() 执行的操作与 [Array.join](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_join.asp" \o "JavaScript join() 方法) 执行的操作是相反的。  如果把空字符串 ("") 用作 separator，那么 stringObject 中的每个字符之间都会被分割。 | | | | | | |
| 找 | charAt(); | stringObject.charAt(index);  //下标从零开始； | | | | | | |
| 找 | indexOf(); | stringObject.indexOf(searchvalue,fromindex);  var str="Hello world!";str.indexOf("Hello")str.indexOf("World")str.indexOf("world")  //0 //-1 //6 | | | | | | |
| lastIndexOf() | var str="shenyonglai"; log(str.lastIndexOf("shen")); log(str.lastIndexOf("Shen")); log(str.lastIndexOf("n")); //0 //-1 //6 | | | | | | |
| 截取 | str.slice(start,end);  str.substring(start,stop);  str.substr(start,length); | | | | | | | |
| String 对象的方法 slice()、substring() 和 substr() （不建议使用）都可返回字符串的指定部分；  substring(+,+) 不接受负的参数,而[slice(+-,+-)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_slice_string.asp" \o "JavaScript slice() 方法) 和 [substr(+-,+)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_substr.asp" \o "JavaScript substr() 方法) 允许负参数； | | | | | | | |
| slice()使用两个字符的位置来指定子串，而 substr() 则用字符位置和长度来指定子串；  还要注意的是，String.slice() 与 Array.slice() 相似； | | | | | | | |
| 如果参数 start 与 stop 相等，那么substring返回的就是一个空串（即长度为 0 的字符串）；  如果 start 比 stop 大，那么该方法在提取子串之前会先交换这两个参数； | | | | | | | |
| substr() 的参数指定的是子串的开始位置和长度，因此它可以替代 substring() 和 slice() 来使用； | | | | | | | |
| 大小写 | toLocaleLowerCase()  toLocaleUpperCase()  toLowerCase()  toUpperCase()  toLocaleLowerCase()和toLocaleUpperCase()方法则是针对特定地区的实现。  对有些地区来说，针对地区的方法与其通用方法得到的结果相同，但少数语言(如土耳其语言)会为Unicode大小写转换应用特殊的规则，这时候就必须使用针对地区的方法来保证实现正确的转换 | | | | | | | |
| 连接字符串 | concat();  不建议使用；  使用"+"即可； | | var str1="123"  var str2="456"  var str3=str1.concat(str2); | | | log(str1.concat(str2)+"<br>");  log(str3+"<br/>");  log(str1); | | 123456 123456 123 |
| 进制切换 | 432.54返回该数字的整数部分？如何转换为八进制和十进制  var fltNumber=432.54; var intNumber=parseInt(fltNumber); var octNumber=intNumber.toString(8); var hexNumber=intNumber.toString(16); | | | | | | | |

js正则表达式之exec()、test()、match()、search()方法

|  |  |
| --- | --- |
|  | var str = "tesst, Tes, tst, tset, Test, Tesyt, sTes";  var reg = /ess/;  console.log(reg.test(str));//true  console.log(reg.exec(str));//[ig'zek] [ 'es', index: 7, input: 'test, Tes, tst, tset, Test, Tesyt, sTes' ]  console.log(str.search(reg));// 1  console.log(str.match(reg));// [ 'es', 'es', 'es', 'es', 'es' ]  console.log(str.indexOf("es"));//1 //不接受正则作为参数  console.log(str.lastIndexOf("es"))//38 |
| 结果：  1、/reg/.exec(str)方法，有一个字符串参数，返回一个数组，数组的第一个条目是第一个匹配；其他的是反向引用。所以第一个返回的结果是第一个匹配的值es（不区分大小写）。  2、str.match()方法返回一个包含在字符串中所有匹配的数据。这个方法调用string对象，同时传给它一个RegExp对象。所以第二个弹出语句返回的是所有符合正则表达式的数组。  3、str.search()的字符串方法与indexOf()有些类似，但是它可以使用一个RegExp对象而非仅仅一个子字符串。  search()方法返回第一个匹配值的位置。所以第三处弹出的是"1"，即第二个字符就匹配了。  注意的是search()方法不支持全局匹配正规表达式（带参数g）。 |
| test | test 方法检查字符串是否与给出的正则表达式模式相匹配，如果是则返回 true，否则就返回 false。 它指出在被查找的字符串中是否匹配给出的正则表达式。  rgexp.test(str)  rgexp ：必选项。包含正则表达式模式或可用标志的正则表达式对象。  str ：必选项。要在其上测试查找的字符串。 |
| 例子：  log(/ab/.test("abc")); //true log(/\w\*ab/.test("abc")); //true log(/\w+ab/.test("abc")); //false |
| 例子：   |  |  | | --- | --- | | **var s** = "I love you"; **var re** = /o/g; **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); **log**(**re**.test(**s**)); | //true //true //false //true //true //false //true | | **var regex** = /\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}/g; **log**(**regex**.test("2012-12-12 12:12:12")); **log**(**regex**.test("2012-12-12 12:12:12")); | //true //false | | **var regex** = /\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}/g; **log**(**regex**.test("2012-12-12 12:12:12")); **regex**.lastIndex = 0; **log**(**regex**.test("2012-12-12 12:12:12")); | //true  //true |   可以看到：对于同一个正则表达式对象regex，不能重复调用：第一次返回true，第二次就返回false，很显然这种效果不是我们想要的。  表达式在匹配全局标志g的时候须注意：RegExp.test()继承正则表达式的lastIndex属性；当第二次调用test()的时候，lastIndex指向下一次匹配所在位置1，所以第二次匹配不成功，lastIndex重新指向0，等于第三次又重新匹配。  解决方法：将test()的lastIndex属性每次重新指向0，re.lastIndex = 0; |
| match | 使用正则表达式模式对字符串执行查找：  1.找不到返回null；  2.非全局匹配正则表达式，和exec()一样，在数组中返回了index 和 input属性，数组的长度为1，考虑存在子表达式的情况  3.全局匹配正则表达式，和exec()方法有很大的不同了，他直接返回了所有符合匹配的子字符串的数组，另外，index和input属性也不在其中了，所以这个方法效率可能会高一些，但是如果你需要更多的信息，则用exec()  找到了返回要找的字符数组（可以使用.length方法来判断共有几个满足条件）  stringObj.match(rgExp)  stringObj ：必选项。字符串  rgExp ：必选项。正则表达式对象、字符串 |
| **var s** = **"The rain in Spain falls mainly in the plain"**; log(**s**.match(/ain/)); // ["ain", index: 5, input: "The rain in Spain falls mainly in the plain"] log(**s**.match(/ain/g));s // ["ain", "ain", "ain", "ain"] |
| search | 返回与正则表达式查找内容匹配的第一个子字符串的位置（偏移位）；如果找到则返回子字符至开始处的偏移位，否则返回-1。  stringObj.search(rgExp)  stringObj ：必选项。要在其上进行查找的 String 对象或字符串文字。  rgExp ：必选项。包含正则表达式模式和可用标志的正则表达式对象。 |
| var s = "The rain in Spain falls mainly in the plain."; log(s.search(/a/i)); //5 log(s.search("a")); //5 log(s.search(/z/i)); //-1 log(s.search("z")); //-1 |
| exec | 用正则表达式模式在字符串中查找，并返回该查找结果的第一个值（数组），如果匹配失败，返回null。  rgExp.exec(str)  rgExp ：必选项。包含正则表达式模式和可用标志的正则表达式对象。  str ：必选项。要在其中执行查找的 String 对象或字符串文字。  返回数组包含：  input：整个被查找的字符串的值；  index：匹配结果所在的位置（位）；  lastInput：下一次匹配结果的位置；  arr：结果值，arr[0]全匹配结果，arr[1,2...]为表达式内()的子匹配，由左至右为1,2...。  说明：  根据手册，exec只返回匹配结果的第一个值，比如上例如果不用while循环，将只返回'I'（尽管i空格后的love和you都符合表达式），无论re表达式用不用全局标记g。但是如果为正则表达式设置了全局标记g，exec 从以 lastIndex 的值指示的位置开始查找。如果没有设置全局标志，exec 忽略 lastIndex 的值，从字符串的起始位置开始搜索。利用这个特点可以反复调用exec遍历所有匹配，等价于match具有g标志。  当然，如果正则表达式忘记用g，而又用循环（比如：while、for等)，exec将每次都循环第一个，造成死循环。exec的输出将包含子匹配项。 |
| 例子1[非全局匹配][全局匹配]：  // 尝试去匹配搜索字符串。返回的数组包含了所有 "ain" 出现的四个匹配 **var s** = **"The rain in Spain falls mainly in the plain"**; log(/ain/.exec(**s**)); // ["ain", index: 5, input: "The rain in Spain falls mainly in the plain"] log((/ain/g).exec(**s**)); // ["ain", index: 5, input: "The rain in Spain falls mainly in the plain"] |
| 例子2：  **var src** = "http://753.blog.163.com/blog/I love you!"; **var re** = /o/g; // 注意g将全文匹配，不加将永远只返回第一个匹配。 **var result**; **while** ((**result** = **re**.exec(**src**)) != **null**) {  **/\*exec使result返回匹配的第一个，while循环一次将使re在g作用寻找下一个匹配。\*/  log**("" + **result**);  **log**(**result**); }  // o // [ 'o', index: 13, input: 'http://753.blog.163.com/blog/I love you!' ] // o // [ 'o', index: 21, input: 'http://753.blog.163.com/blog/I love you!' ] // o // [ 'o', index: 26, input: 'http://753.blog.163.com/blog/I love you!' ] // o // [ 'o', index: 32, input: 'http://753.blog.163.com/blog/I love you!' ] // o // [ 'o', index: 37, input: 'http://753.blog.163.com/blog/I love you!' ] |
| 例子3（子匹配项）：  **var s** = "The rain in Spain falls mainly in the plain"; **var r** = /[\w]\*(ai)n/ig.exec(**s**); **log**(**r**); **for** (**var key in r**) {  **log**(**key** + "-" + **r**[**key**]); } //[ 'rain', 'ai', index: 4, input : 'The rain in Spain falls mainly in the plain' ]  //0-rain //1-ai //index-4 //input-The rain in Spain falls mainly in the plain |
| 例子4（字匹配项）：  var result = /(\d+)-(\w+)/.exec('12-ab');  // ["12-ab", "12", "ab", index: 0, input: "12-ab"]  //为什么上面返回的"数组"是那么奇怪，按照[w3school][2]的说法就是：exec() 都会把完整的细节添加到它返回的数组中，这里的细节指的就是index和input  整个正则表达式匹配的文本："12-ab"  第一个子表达式匹配的文本："12"  第二个子表达式匹配的文本："ab" |
|  | 例子4（与match）：  var s = "The rain in Spain falls mainly in the plain";  console.log(/ain/.exec(s));  console.log(s.match(/ain/));  console.log(/ain/g.exec(s));  console.log(s.match(/ain/g));  // [ 'ain', index: 5, input: 'The rain in Spain falls mainly in the plain' ]  // [ 'ain', index: 5, input: 'The rain in Spain falls mainly in the plain' ]  // [ 'ain', index: 5, input: 'The rain in Spain falls mainly in the plain' ]  // [ 'ain', 'ain', 'ain', 'ain' ] |
| split 方法 | |
|  | 将一个字符串分割为子字符串，然后将结果作为字符串数组返回。 在 stingObj 中每个出现 separator 的位置都要进行分解。separator 不作为任何数组元素的部分返回。  stringObj.split([separator[, limit]])  stringObj:必选项。要被分解的 String 对象或文字。该对象不会被 split 方法修改。  separator:可选项。字符串或 正则表达式 对象，它标识了分隔字符串时使用的是一个还是多个字符。如果忽略该选项，返回包含整个字符串的单一元素数组。  limit:可选项。该值用来限制返回数组中的元素个数 。 |
|  | **var str** = "The rain in Spain falls mainly in the plain.";  **console**.log(**str**.split(/s/i)); //正则表达式，用不分大不写的s进行分隔。 //[ 'The rain in ', 'pain fall', ' mainly in the plain.' ]  **console**.log(**str**.split(/\s+/)); //空格切分，注意与 .split(" ") 的区别 //[ 'The', 'rain', 'in', 'Spain', 'falls', 'mainly', 'in', 'the', 'plain.' ]  **console**.log(**str**.split("")); //[ 'T', 'h', 'e', ' ', 'r', 'a', 'i', 'n', ' ', 'i', 'n', ' ', 'S', 'p', 'a', 'i', 'n', ' ', 'f', 'a', 'l', 'l', 's', ' ', ' ', ' ', ' ', 'm', 'a', 'i', 'n', 'l', 'y', ' ', 'i', 'n', ' ', 't', 'h', 'e', ' ', 'p', 'l', 'a', 'i', 'n', '.']  **console**.log(**str**.split(" ")); //[ 'The', 'rain', 'in', 'Spain', 'falls', '', '', '', 'mainly', 'in', 'the', 'plain.' ]  **console**.log(**str**.split("",3));  //限制 //[ 'T', 'h', 'e' ] |

replace 方法 [不会改变原字符]

|  |  |
| --- | --- |
| 简介 | 用于在字符串中用一些字符替换另一些字符，或替换一个与正则表达式匹配的子串；返回替换后字符串的复制。  string.replace(subStr/reg,replaceStr/function)  参数：  pattern: [字符串][RegExp]  replacement： [替换匹配项的字符串][处理函数的返回值] |
|  | 1. replacement 为字符串:   1.当未找到匹配项的时候，返回原始字符串:  'aaaa'.replace('bbb', 'b') //"aaaa"  2.当pattern为字符串或者为非全局的RegExp对象的时候，只替换找到的第一项匹配项:  'aaaa'.replace('a', 'b') //"baaa"  'aaaa'.replace(/\w/, 'b') //"baaa"  3.当pattern为全局的RegExp对象的时候，替换每一项匹配项:  'aaaa'.replace(/\w/g, 'b') //"bbbb" |
| 1. replacement为函数时：   'aaaa'.replace(/\w/g, function() {  return 'b';  });  // "bbbb"  'aaaa'.replace(/\w/g, function(value) {  return value.toUpperCase();  });  // "AAAA"  结论：  函数的返回值将作为替换字符串  函数的第一个参数的值是每一个匹配项,当然还有第二个参数，它的值是每个匹配项在原始字符串的中位置，从0开始 |
|  | **var str**="jaajjejaaisAAdjAAjeji"; **str**.replace("aa","--");//j--jjejaaisAAdjAAjeji **str**.replace(/aa/g,"--");//j--jjej--isAAdjAAjeji **str**.replace(/aa/i,"--");//j--jjejaaisAAdjAAjeji **str**.replace(/aa/i,"--");//j--jjej--is--dj--jeji |
| $符号 | 1. replacement 中 $ 字符具有特定的含义  |  |  | | --- | --- | | 字符 | 替换文本 | | $1、...、$99 | 与 regexp 中的第 1 到第 99 个子表达式相匹配的文本。 | | $& | 与 regexp 相匹配的子串 | | $` | 位于匹配子串左侧的文本 | | $' | 位于匹配子串右侧的文本 | | $$ | 直接量符号 |   //第一种情况：  'aa11AA'.replace(/([a-z]+)(\d+)([A-Z]+)/g, '$1'); // "aa"  'aa11AA'.replace(/([a-z]+)(\d+)([A-Z]+)/g, '$2'); // "11"  'aa11AA'.replace(/([a-z]+)(\d+)([A-Z]+)/g, '$3'); // "AA"  'aa11AA'.replace(/([a-z]+)(\d+)([A-Z]+)/g, '$4'); // "$4"  //猜想 如果是 $4 回事什么呢？ undefined ? 要是没有该子项，则当成普通字符串处理了  //第二种情况：'aa11AA'.replace(/([a-z]+)(\d+)([A-Z]+)/g, '$&'); //"aa11AA"  //第三种情况：'aa11AA'.replace(/(\d+)/g, '$`'); //"aaaaAA"  //第四种情况：'aa11中国'.replace(/(\d+)/g, "$'"); //"aa中国中国"  //第五种情况：'aa11AA'.replace(/(\d+)/g, '$$'); //"aa$AA" |
| **var str** = "today: 2011-03-29"; **var resStr** = **str**.replace(/([\d]{4})-([\d]{1,2})-([\d]{1,2})/, **function** ($0, $1, $2, $3) {  **log**($0 + " | " + $1 + " | " + $2 + " | " + $3 + " | ");  **return** ($2 + "/" + $1); }); **log**(**resStr**);  //2011-03-29 | 2011 | 03 | 29 | //today: 03/2011 |
| **var str** = "today: 2011-03-29"; **var resStr** = **str**.replace(/([\d]{4})-([\d]{1,2})-([\d]{1,2})/, **function** (a, b, c, d) {  **log**(a + " | " + b + " | " + c + " | " + d + " | ");  **return** (c + "/" + b); }); **log**(**resStr**);  //2011-03-29 | 2011 | 03 | 29 | //today: 03/2011 |
| // 华氏温度可能模式有：123F或123.4F。注意，这里用了g模式 **var str** = "Water: 32.2F Oil: 20.30F Other: 12F"; **var strRes** = **str**.replace(/(\d+(\.\d\*)?)F\b/g, **function** ($1, $2, $3, $4, $5) {  **log**($1 + " | " + $2 + " | " + $3 + " | " + $4 + " | " + $5);  //[匹配结果,第一匹配项,第二匹配项,位置,,,输入]  **return** "----"; }); **log**(**strRes**); //32.2F | 32.2 | .2 | 7 | Water: 32.2F Oil: 20.30F Other: 12F //20.30F | 20.30 | .30 | 18 | Water: 32.2F Oil: 20.30F Other: 12F //12F | 12 | undefined | 32 | Water: 32.2F Oil: 20.30F Other: 12F //Water: ---- Oil: ---- Other: ---- |
| // 说明华氏温度可能模式有：123F或123.4F **var strTemp** = "Water: 32.2F and Oil: 20.30F.".replace(/(\d+(\.\d\*)?)F\b/g, **function** (a, b) {  **log**(a + " | " + b);  **return** ( b - 32 + "C"); });  log(**strTemp**);  //32.2F | 32.2 //20.30F | 20.30 //Water: 0.20000000000000284C and Oil: -11.7C. |

demo

|  |  |  |
| --- | --- | --- |
|  |  | |
| 改变顺序 | 置换[1个] | **var name** = '"a"-"b"-"c"'; **log**(**name**.replace(/"(\w\*)"/g, "'$1'"));  //'a'-'b'-'c' | |
| 置换[2个] | **var name**='Steven Nancy'; **log**(**name**.replace(/(\w+)\s\* \s\*(\w+)/, "$2 $1")); //Nancy Steven | |
| 置换[3个]  3 2 1 6 5 4 | **/\* (\d)中（）不能少，固定格式，子匹配项 \*/ var s**="1 2 3 4 5 6 "; **var ss**=**s**.replace(/(\d)\s\* \s\*(\d)\s\* \s\*(\d)/g,"$3 $2 $1"); **log**(**ss**); //3 2 1 6 5 4 | |
| 首字母大写：  Aaa Bbb Ccc | **var name** = 'aaa bbb ccc';  **var nameA**=**name**.replace(/\b\w+\b/g, **function** (a) {  **return** a.substring(0, 1).toUpperCase() + a.substring(1); });  **var nameB**=**name**.replace(/(\w)\w\*\b/g, **function** (a, b) {  **return** b.toUpperCase() + a.substring(1); });  **console**.log(**name**); **console**.log(**nameA**); **console**.log(**nameB**);  //aaa bbb ccc //Aaa Bbb Ccc //Aaa Bbb Ccc | |
| 123.456.789 | var a="123456789";  var b=a.replace(/\w{3}/g,function(aa){  return aa.concat("."); });  log(b.slice(0, b.length-1)) | |
| 01.01.20.10 | "001.002.200.000.202.200.000".replace(/(\.|^)([0]+)(\d)/g, "$1$3")  > 可用于ip地址去零 | |
| 12.345.678.913.355 | **var a**=**"12345678913355"**; **var b**=**a**.split(**""**).reverse().join(**""**).replace(/\w{3}/g,**function**(aa){  **return** aa.concat(**"."**); }); **var c**=**b**.split(**""**).reverse().join(**""**); | |
| 替换一段话中的某个单词 | **var strToSearch**=**"The fun of function is that they are functional."**; **var afterMatch**=**strToSearch**.replace(/\bfun\b/,**"-----"**);  //The ----- of function is that they are functional. | |
| HTML格式化 | 替换字符串中得HTML字符（把&quot;替换为",&lt;替换为<，&gt;替换为>）  String.prototype.deentityfy = function() {  var entity = {  quot: '"',  lt: '<',  gt: '>'  };  return function() {  return this.replace(/&([^&;]+);/g, function(a, b) {  var r = entity[b];  return typeof r === 'string' ? r : a;  }); //end of replace  }; } ();  分析：  for (var i = 0; i < arguments.length; i++) {  log(arguments[i] + '<br/>'); } | |
| 获取函数的名字 | function fun\_name (num){  var tmp = arguments.callee.toString();  var re = /function\s\*(\w\*)/i;  var matches = re.exec(tmp);  log(matches[1]);  }  fun\_name(); | |
| 表单验证JS | **window**.onload=**function**(){  **var** btn=**document**.getElementById("myform");  btn.onsubmit=**function**(){  **return checkAll**();  } }; **function checkAll**(){  **var** pass=**document**.getElementById("pwd").value;  **var** email=**document**.getElementById("email").value;  **var** flag=**false**;  **if**(email==""||pass.length<6||email.indexOf("@")==-1||email.indexOf(".")==-1){  **log**("有误")  }**else**{  flag=**true**;  }  **return** flag; } | |
| 'border-bottom-color' --> borderBottomColor | var str = 'border-bottom-color';  /\*这里是常规方法\*/  function transform(str) {  var arr = str.split('-'); //[border,bottom,color]  for (var i = 1; i < arr.length; i++) {  arr[i] = arr[i].charAt(0).toUpperCase() + arr[i].substring(1); //[border,Bottom,Collor]  }  return arr.join('');  }  console.log(transform(str)); | |
| var str = 'border-bottom-color';  /\*这里是正则方法\*/  function transform(str) {  //()代表正则中的第一个子项 $0代表正则的整体 $1代表正则中的第一个子项  return str.replace(/-(\w)/g, function ($0, $1) {  return $1.toUpperCase();  });  }  console.log(transform(str)); | |
| 添加千位符 | var str = '12345432'; //->[]  function spli(str) {  var iNum = str.length % 3; //->0,1,2  var prev = '';  var arr = [];  var iNow = 0;  var tmp = '';  //千分号是从后向前三位三位加的，所以把三位余出来的从前面提取到  if (iNum != 0) {  prev = str.substring(0, iNum);  arr.push(prev);  }  //取到多余部分后面的  str = str.substring(iNum);  for (var i = 0; i < str.length; i++) {  iNow++;  tmp += str[i];  if (iNow == 3 && tmp) {  //数组里面放的是三位的数  arr.push(tmp);  tmp = '';  iNow = 0;  }  }  return arr.join(',');  }  console.log(spli(str)); | |
| var str = '12345432';  function spli(str) {  var re = /(?=(?!\b)(\d{3})+$)/g;  return str.replace(re, ',');  }  console.log(spli(str)); | |
| (?=) :前向声明  (?!) :反前向声明  //向某个符合要求的片段前加字符  var str = 'abacad';  var re = /(?=a)/g;  str = str.replace(re,'\*');  console.log(str);//\*ab\*ac\*ad | |
| 返回只包含数字类型的数组 | var str = 'abc234koi45jodjnvi789'; //得到[234,45,789]  var re = /(\d+)/g;  var arr = [];  str.replace(re, function ($0, $1) {  arr.push(parseInt($1));  });  console.log(arr);//[ 234, 45, 789 ] | |
| var str = 'abc234koi45jodjnvi789';  console.log(str.split(/[a-z]+/g));//[ '', '234', '45', '789' ] | |
| 正则分组 | var testStr = `<div>  <img src='/a.jpg' alt='' />  <span>test</span>  <img src='/b.jpg' alt='' />  <span>test</span>  <img src='/c.png' alt='' />  </div>`;  var reg = /<img\ssrc='(.\*?)'\s+alt=''\s\*\/>/g;  var match = reg.exec(testStr);  var results = [];  while (match != null) {  results.push(match[1]);  match = reg.exec(testStr);  }  console.log(results);//["/a.jpg", "/b.jpg", "/c.png"] | |
| results = [];  testStr.replace(/<img\ssrc='(.\*?)'\s+alt=''\s\*\/>/g,function($0,$1){  results.push($1)  })  console.log(results);//["/a.jpg", "/b.jpg", "/c.png"] | |
|  |  | |
|  |  | |

正则

|  |  |
| --- | --- |
|  | js正则表达式常用函数方法及简单应用  <http://www.haorooms.com/post/js_RegExp_fn_use>  重新拎一遍js的正则表达式  <http://www.haorooms.com/post/js_regex_refuse>  正则表达式研究系列(17)  <http://www.qdfuns.com/house/17398/note/class/id/995a01d28bda579571d0c3239ac96bbf.html>  [59分钟学会正则表达式](https://www.jeffjade.com/2014/12/25/2014-12-25-regular-expression/" \o "59分钟学会正则表达式)  <https://www.jeffjade.com/2014/12/25/2014-12-25-regular-expression/> |
| \b | \b是正则表达式规定的一个特殊代码（好吧，某些人叫它元字符，metacharacter），代表着单词的开头或结尾，也就是单词的分界处。虽然通常英文的单词是由空格，标点符号或者换行来分隔的，但是\b并不匹配这些单词分隔字符中的任何一个，它只匹配一个位置"  问题来了，什么叫只匹配一个位置？？？？    要知道是一个单词的开头，这个\b是去匹配开头的分隔符还是匹配开头的第一个字母？？  "如果需要更精确的说法，\b匹配这样的位置：它的前一个字符和后一个字符不全是(一个是,一个不是或不存在)\w（匹配字母或数字或下划线或汉字）"    这 句话很隐晦的说出了是去匹配第一个字母，而不是去匹配分割符（我这说的匹配分隔符和上边说的不去匹配任何一个分隔符并不矛盾，因为我经过实验已经进本猜出 了一个单词两边只要没有数字、字母、汉字、下划线就可以认为是独立单词，我说的匹配分隔符是去判断单词前一个字符是不是在这四个之外，是的话就认为是单词 开头）    再加上下边的两个例子就可以很清晰的理解这句很隐晦的话。  引用Microsoft   VBScript   Regular   Expression  Dim myReg As RegExp  Set myReg = New RegExp    myReg.Pattern = "a\b.\*h.\*"  Print myReg.Test("a,chia")  输出的结果为Ture 说明是匹配的a，而不是，  Dim myReg As RegExp  Set myReg = New RegExp    myReg.Pattern = "a.\* \b h.\*"  Print myReg.Test("a,chia")  输出的结果为flash 说明匹配的是h，而不是c    这两个例子非常有助于理解\b的真实含义，希望对大家有帮助！    接下来就是我保证目前网络中没有的内容了！！！！还有两点需要说明：  1、\b只能匹配字母、数字、汉字、下划线  2、\b就近匹配，比如\bAB 匹配A，AB\b 匹配B；但如果只写一个\b或者在两个字母、数字、汉字、下划线之间有\b时就在所有字符或者两个字母、数字、汉字、下划线之间所有字符去逐个匹配  相信如果你能看懂我上面所写的东西，对于\b的使用算是没有任何问题了！对正则表达式的学习有一个好的开始和基础！ |
|  | 非捕获性分组释疑  在看人民邮电出版社出版的《JavaScript高级程序设计》  看到7.3.4 非捕获性分组这里，书上说非捕获性分组不会创建反向引用，下面是例子：  var sToMatch = "#123456789";  var reNumbers = /#(?:\d+)/;  reNumbers.test(sToMatch);  log(RegExp.$1);//空    var sToMatch = "en-us rv:0.9.4";  var reNumbers = /rv:(\d+\.\d+(?:\.\d+)?)/;  var result = reNumbers.test(sToMatch);  log(RegExp.$1);    这里log输出就成了0.9.4  最后那个.4是非捕获性分组捕获到的  这里就奇怪了，为什么前面的非捕获性分组不能捕获，而嵌套的非捕获性分组就又可以捕获了？  而且这里好像不用非捕获性分组照样可以达到相同的目的，为什么要用非捕获性分组呢？        要了解非捕获组就要先了解捕获组，之后再了解为什么会有非捕获组的出现  简单点说，捕获组就是把(Expression)中匹配到的内容保存到一个按"("出现的顺序编号的组里，以供后续引用，引用的方式有反向引用，或是RegExp.$number等方式，不同的语言，支持的引用方式不同  只要使用了()，默认为使用了捕获组，而这就带来一个问题，有些场景不得不使用()，但又不关心它匹配到的内容，比如写一个匹配24小时制HH:mm:ss的时间的正则如下  ([01][0-9]|2[0-3])(:([0-5][0-9])){2}  通常关心的只是整体的时间，并不关心局部的内容，这样就产生了一种副作用，将不关心的内容单独保存到内存中，只会浪费资源，降低效率  非捕获组就是为了抵消这一副作用来产生的，非捕获组只参与匹配，但不会把匹配到的内容捕获到组里  所以非捕获组根本就不参与编号，也就无从谈起它对应哪个$number  在取不存在的编号的捕获组时，有些语言会返回空字符串，有些语言会报异常  (\d+\.\d+(?:\.\d+)?)中，整体是一个捕获组，按"("出现的顺序，编号为1，(?:\.\d+)虽然是非捕获组，也是要参与匹配的，只是不将匹配结果单独保存到组里而已  还需要说明的是，在绝大多数语言中，正则表达式整体对应的是$0，捕获组的编号是从1开始的  在有些语言中，还支持(?<name>Expression)的命令捕获组语法，所以有以下两种语法属于捕获组  (Expression)  (?<name>Expression)  其余的(?...)之类的语法定义的字符序列都不属于捕获组 |

## array

|  |  |  |
| --- | --- | --- |
|  |  | |
| API | new | var arrayObj = new Array();  创建一个数组;  var arrayObj = new Array([size]);  创建一个数组并指定长度，注意不是上限，是长度;  var arrayObj = new Array([element0[, element1[, ...[, elementN]]]]);  创建一个数组并赋值;  要说明的是，虽然第二种方法创建数组指定了长度，但实际上所有情况下数组都是变长的，也就是说即使指定了长度为5，仍然可以将元素存储在规定长度以外的，注意：这时长度会随之改变。 |
| i | var testGetArrValue=arrayObj[1]; //获取数组的元素值  arrayObj[1]= "这是新值"; //给数组元素赋予新的值 |
| 添加 | arrayObj. push([item1 [item2 [. . . [itemN ]]]]);// 将一个或多个新元素添加到数组结尾，并返回数组新长度  arrayObj.unshift([item1 [item2 [. . . [itemN ]]]]);// 将一个或多个新元素添加到数组开始，数组中的元素自动后移，返回数组新长度 |
| 删除 | arrayObj.pop(); //移除最后一个元素并返回该元素值  arrayObj.shift(); //移除最前一个元素并返回该元素值，数组中元素自动前移  arrayObj.splice(deletePos,deleteCount); //删除从指定位置deletePos开始的指定数量deleteCount的元素，数组形式返回所移除的元素 |
| 截取  合并 | arrayObj.slice(start, [end]); //以数组的形式返回数组的一部分，注意不包括 end 对应的元素，如果省略 end 将复制 start 之后的所有元素  arrayObj.concat([item1[, item2[, . . . [,itemN]]]]); //将多个数组（也可以是字符串，或者是数组和字符串的混合）连接为一个数组，返回连接好的新的数组 |
| 拷贝 | arrayObj.slice(0); //返回数组的拷贝数组，注意是一个新的数组，不是指向  arrayObj.concat(); //返回数组的拷贝数组，注意是一个新的数组，不是指向 |
| 排序 | arrayObj.reverse(); //反转元素（最前的排到最后、最后的排到最前），返回数组地址  arrayObj.sort(); //对数组元素排序，返回数组地址 |
| 字符串化 | arrayObj.join(separator); //返回字符串，这个字符串将数组的每一个元素值连接在一起，中间用 separator 隔开。  toLocaleString 、toString 、valueOf：可以看作是join的特殊用法，不常用 |
|  | arrayObj.splice(insertPos,0,[item1[, item2[, . . . [,itemN]]]]);//将一个或多个新元素插入到数组的指定位置，插入位置的元素自动后移，返回""。 |
| 属性 | length | Length属性表示数组的长度，即其中元素的个数。因为数组的索引总是由0开始，所以一个数组的上下限分别是：0和length-1。和其他大多数语言不同的是，JavaScript数组的length属性是可变的，这一点需要特别注意。当length属性被设置得更大时，整个数组的状态事实上不会发生变化，仅仅是length属性变大；当length属性被设置得比原来小时，则原先数组中索引大于或等于length的元素的值全部被丢失。下面是演示改变length属性的例子：  var arr=[12,23,5,3,25,98,76,54,56,76];  //定义了一个包含10个数字的数组  log(arr.length); //显示数组的长度10  arr.length=12; //增大数组的长度  log(arr.length); //显示数组的长度已经变为12  log(arr[8]); //显示第9个元素的值，为56  arr.length=5; //将数组的长度减少到5，索引等于或超过5的元素被丢弃  log(arr[8]); //显示第9个元素已经变为"undefined"  arr.length=10; //将数组长度恢复为10  log(arr[8]); //虽然长度被恢复为10，但第9个元素却无法收回，显示"undefined"  由上面的代码我们可以清楚的看到length属性的性质。 |
|  |  | 元素的数量不等同于其长度  一般情况下，当我们在谈到数组长度的时候，我们会认为其长度要么是数组值的数量，要么是我们手动给数组设定的长度。但是事实上，数组的长度取决于其内部最大的已经存在的索引。  长度是一个非常灵活的属性。无论你是否曾实现调整了数组的长度，只要你不断的给数组添加新的值，它的长度也会随之增长。  但length对象不仅可以显式的设置，它也有可能被隐式修改。JavaScript中可以使用一个未声明过的变量，同样，也可以使用一个未定义的数组元素（指索引超过或等于length的元素），这时，length属性的值将被设置为所使用元素索引的值加1。例如下面的代码：  var arr=[12,23,5,3,25,98,76,54,56,76];  log(arr.length);  arr[15]=34;  log(arr.length);  代码中同样是先定义了一个包含10个数字的数组，通过log语句可以看出其长度为10。随后使用了索引为15的元素，将其赋值为15，即arr[15]=34，这时再用log语句输出数组的长度，得到的是16。无论如何，对于习惯于强类型编程的开发人员来说，这是一个很令人惊讶的特性。事实上，使用new Array()形式创建的数组，其初始长度就是为0，正是对其中未定义元素的操作，才使数组的长度发生变化。  由上面的介绍可以看到，length属性是如此的神奇，利用它可以方便的增加或者减少数组的容量。因此对length属性的深入了解，有助于在开发过程中灵活运用。  在上面的例子中，你可以看到我给数组的索引15只指定了一个值，之后长度变成了16。  现在，如果你觉得给index 15添加一个值，数组就会自动创建索引0-15，那么你的推测就出现了错误。数组中并没有应经存在的索引0-15。你可以使用in operator来查看。  var ary = [];  ary.length = 3;  console.log(ary.length); //3  ary[5] = "abcd";  console.log(ary.length); //6  console.log(0 in ary); //false  上面的ary数组被我们成为稀疏数组（sparse array），这个数组的索引不会持续的被创建，而且索引之间有空气。sparse数组的对立面为密集数组（dense array）。密集数组的索引会被持续的创建，其元素的数量等于其长度。  数组的长度属性也可以用来缩短数字，确保数组中索引的最大数量永远小于数组本身，因为在默认情况下，长度的数值永远会大于索引数量的最高值。  在下面的例子中，你可以看到，我利用减少ary数组长度的方式，置空了索引5中的元素。  var ary = [];  ary.length = 3;  console.log(ary.length);//3  ary[5] = "abcd";  console.log(ary.length);//6  ary.length = 2;  console.log(ary.length);//2  console.log(ary[5]);//undefined |
|  | prototype | 返回对象类型原型的引用。prototype 属性是 object 共有的。  objectName.prototype  objectName 参数是object对象的名称。  说明：用 prototype 属性提供对象的类的一组基本功能。 对象的新实例"继承"赋予该对象原型的操作。  对于数组对象，以以下例子说明prototype 属性的用途。  给数组对象添加返回数组中最大元素值的方法。要完成这一点，声明一个函数，将它加入 Array.prototype， 并使用它。  function array\_max( )  {     var i, max = this[0];     for (i = 1; i < this.length; i++)     {     if (max < this[i])     max = this[i];     }     return max;  }  Array.prototype.max = array\_max;  var x = new Array(1, 2, 3, 4, 5, 6);  var y = x.max( );  该代码执行后，y 保存数组 x 中的最大值，或说 6。 |
|  | constructor | 表示创建对象的函数。  object.constructor //object是对象或函数的名称。  说明：constructor 属性是所有具有 prototype 的对象的成员。它们包括除 Global 和 Math 对象以外的所有 JScript 固有对象。constructor 属性保存了对构造特定对象实例的函数的引用。  例如：  x = new String("Hi");  if (x.constructor == String) // 进行处理（条件为真）。  或  function MyFunc {  // 函数体。  }  y = new MyFunc;  if (y.constructor == MyFunc) // 进行处理（条件为真）。 |

es5

|  |  |  |
| --- | --- | --- |
| 简述 | <http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/>  <http://www.cnblogs.com/song-song/p/5329374.html> | |
| [forEach](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "foreach" \o "forEach) (js v1.6)  [map](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "map" \o "map) (js v1.6)  [filter](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "filter" \o "filter) (js v1.6)  [some](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "some" \o "some) (js v1.6)  [every](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "every" \o "every) (js v1.6)  [indexOf](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "indexof" \o "indexOf) (js v1.6)  [lastIndexOf](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "lastindexof" \o "lastIndexOf) (js v1.6)  [reduce](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "reduce" \o "reduce) (js v1.8)  [reduceRight](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "reduceright" \o "reduceRight) (js v1.8) | Opera 11+  Firefox 3.6+  Safari 5+  Chrome 8+  Internet Explorer 9+ |
| IE6-IE8浏览器，Array原型扩展可以实现以上全部功能，例如forEach方法:  if (typeof Array.prototype.forEach != "function") {  Array.prototype.forEach = function () {  /\* 实现 \*/  }; | |
| 2个索引方法：indexOf() 和 lastIndexOf()  5个迭代方法：forEach()、map()、filter()、some()、every()  2个归并方法：reduce()、reduceRight() | |
| forEach | **arr**.forEach(**function** (value, index, array) {}) $.each([], **function**(index, value, array) {// 与此相反 }); | |
| 更进一步，forEach除了接受一个必须的回调函数参数，还可以接受一个可选的上下文参数（改变回调函数里面的this指向）（第2个参数）。  array.forEach(callback,[ thisObject])  例子更能说明一切：  var database = {  users: ["张含韵", "江一燕", "李小璐"],  sendEmail: function (user) {  if (this.isValidUser(user)) {  console.log("你好，" + user);  } else {  console.log("抱歉，"+ user +"，你不是本家人");  }  },  isValidUser: function (user) {  return /^张/.test(user);  }  };  // 给每个人法邮件  database.users.forEach( // database.users中人遍历  database.sendEmail, // 发送邮件  database // 使用database代替上面标红的this  );  // 结果：  // 你好，张含韵  // 抱歉，江一燕，你不是本家人  // 抱歉，李小璐，你不是本家  如果这第2个可选参数不指定，则使用全局对象代替（在浏览器是为window），[严格模式](http://dmitrysoshnikov.com/ecmascript/es5-chapter-2-strict-mode/)下甚至是undefined. | |
| 另外，forEach不会遍历纯粹"占着官位吃空饷"的元素的，例如下面这个例子：  var array = [1, 2, 3];  delete array[1]; // 移除 2  log(array); // "1,,3"  log(array.length); // but the length is still 3  array.forEach(log); // 弹出的仅仅是1和3 | |
| 综上全部规则，我们就可以对IE6-IE8进行仿真扩展了，如下代码：  // 对于古董浏览器，如IE6-IE8  if (typeof Array.prototype.forEach != "function") {  Array.prototype.forEach = function (fn, context) {  for (var k = 0, length = this.length; k < length; k++) {  if (typeof fn === "function" && Object.prototype.hasOwnProperty.call(this, k)) {  fn.call(context, this[k], k, this);  }  }  };  } | |
| [map](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "map" \o "map) | array.map(callback,[ thisObject]);  callback的参数也类似：  [].map(function(value, index, array) {  // ...  });  数值项求平方：  var data = [1, 2, 3, 4];  var arrayOfSquares = data.map(function (item) {  return item \* item;  });  log(arrayOfSquares); // 1, 4, 9, 16  callback需要有return值，如果没有，数组所有项都被映射成了undefined； | |
| 在实际使用的时候，我们可以利用map方法方便获得对象数组中的特定属性值们。例如下面这个例子（之后的兼容demo也是该例子）：  var users = [  {name: "张含韵", "email": "zhang@email.com"},  {name: "江一燕", "email": "jiang@email.com"},  {name: "李小璐", "email": "li@email.com"}  ];  var emails = users.map(function (user) { return user.email; }); | |
| Array.prototype扩展可以让IE6-IE8浏览器也支持map方法：  if (typeof Array.prototype.map != "function") {  Array.prototype.map = function (fn, context) {  var arr = [];  if (typeof fn === "function") {  for (var k = 0, length = this.length; k < length; k++) {  arr.push(fn.call(context, this[k], k, this));  }  }  return arr;  };  } | |
| [filter](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "filter" \o "filter) | filter为"过滤"、"筛选"之意。指数组filter后，返回过滤后的新数组。用法跟map极为相似：  array.filter(callback,[ thisObject]); | |
| filter的callback函数需要返回布尔值true或false. 可能会疑问，一定要是Boolean值吗？我们可以简单测试下嘛，如下：  var data = [0, 1, 2, 3];  var arrayFilter = data.filter(function(item) {  return item;  });  console.log(arrayFilter); // [1, 2, 3]  有此可见，返回值只要是弱等于== true/false就可以了，而非非得返回 === true/false | |
| 因此，我们在为低版本浏览器扩展时候，无需关心是否返回值是否是纯粹布尔值（见下黑色代码部分）：  if (typeof Array.prototype.filter != "function") {  Array.prototype.filter = function (fn, context) {  var arr = [];  if (typeof fn === "function") {  for (var k = 0, length = this.length; k < length; k++) {  fn.call(context, this[k], k, this) && arr.push(this[k]);  }  }  return arr;  };  } | |
| map和filter会返回一个新的数组  var newArr = arr.map(function(value,index,thisArray){  //三个参数，分别为索引，值，当前array  return value\*2;  });  var newArr = arr.filter(function(value,index,thisArray){  //三个参数，分别为索引，值，当前array  return value>100; //根据return为false或true来决定是否留下当前元素  }); | |
| [some](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "some" \o "some)  [every](http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/" \l "every" \o "every) | some意指"某些"，指是否"某些项"合乎条件。  every表示是否"每一项"都要靠谱。  用法如下：  array.some(callback,[ thisObject]);  array.every(callback,[thisObject]);  例如下面的简单使用：  //数组中是否有比7大的 var isHaveLagerThan10 = [5, 8, 3, 10].some(function (n) {  return n > 7; }) console.log(isHaveLagerThan10);  我们自然可以使用forEach进行判断，不过，相比some, 不足在于，some只有有true即返回不再执行了。  而且值得注意的是，some方法会在数组中任一项执行函数返回true之后，不在进行循环。 | |
| //数组每个元素是否都大于2 var isHaveLagerThan10 = [5, 8, 3, 10].every(function (n) {  return n > 2; }) console.log(isHaveLagerThan10);//true | |
| indexOf  lastIndexOf | array.indexOf(searchElement[, fromIndex])  返回整数索引值，如果没有匹配（严格匹配），返回-1. fromIndex可选，表示从这个位置开始搜索，若缺省或格式不合要求，使用默认值0，我在FireFox下测试，发现使用字符串数值也是可以的，例如"3"和3都可以。 | |
| lastIndexOf方法与indexOf方法类似：  array.lastIndexOf(searchElement[, fromIndex])  只是lastIndexOf是从字符串的末尾开始查找，而不是从开头。还有一个不同就是fromIndex的默认值是array.length - 1而不是0. | |
| **var arr** = [1, 7, 5, 7, 1, 3];  **log**(**arr**.indexOf(7)); // 1 参数2缺省, 从第一项开始查找 **log**(**arr**.indexOf(4)); // -1 (未找到) **log**(**arr**.indexOf("7")); // -1 (未找到，因为5 !== "5") **log**(**arr**.indexOf(7, 's')); // 1 格式不正确, 从第一项开始查找（'s'被忽略） **log**(**arr**.indexOf(7, 2)); // 3　　　从第三个项之后开始查找 **log**(**arr**.indexOf(1, "3")); // 4 (从3号位开始搜索) **log**("---------------------------------------------------"); **log**(**arr**.lastIndexOf(7)); // 3 缺省, 从末尾第一项开始查找 **log**(**arr**.lastIndexOf(4)); // -1 (未找到) **log**(**arr**.lastIndexOf ('7')); // -1 未找到, 返回-1 **log**(**arr**.lastIndexOf(7, 3)); // 3 (从后往前，索引值小于3的开始搜索) **log**(**arr**.lastIndexOf (7, 2)); // 1 　　从末尾第三项往前查找 **log**(**arr**.lastIndexOf (7, 's')); // -1 格式不正确, 从末尾第一项开始查找 | |
|  | **var arr** = [  {a:"a"},  {a:"b"},  {a:"c"} ]; **log**(**arr**.indexOf({a: "a"})); //-1 | |
| reduce | reduce是JavaScript 1.8中才引入的，中文意思为"减少"、"约简"；功能上更接近于"迭代"、"递归(recursion)"  array.reduce(callback[, initialValue])  callback函数接受4个参数：之前值、当前值、索引值以及数组本身。  initialValue参数可选，表示初始值。若指定，则当作最初使用的previous值；如果缺省，则使用数组的第一个元素作为previous初始值，同时current往后排一位，相比有initialValue值少一次迭代。   1. 返回一个累加值：   var sum = [1, 2, 3, 4].reduce(function (previous, current, index, array) {  return previous + current;  });  console.log(sum); // 10   1. 说明：   因为initialValue不存在，因此一开始的previous值等于数组的第一个元素。  从而current值在第一次调用的时候就是2.  最后两个参数为索引值index以及数组本身array.  以下为循环执行过程：  // 初始设置  previous = initialValue = 1, current = 2  // 第一次迭代  previous = (1 + 2) = 3, current = 3  // 第二次迭代  previous = (3 + 3) = 6, current = 4  // 第三次迭代  previous = (6 + 4) = 10, current = undefined (退出)   1. 有了reduce，我们可以轻松实现二维数组的扁平化：   var matrix = [  [1, 2],  [3, 4],  [5, 6]  ];  var flatten = matrix.reduce(function (previous, current) {  return previous.concat(current);  });  console.log(flatten); // [1, 2, 3, 4, 5, 6] | |
| reduceRight | reduceRight跟reduce相比，用法类似：  array.reduceRight(callback[, initialValue])  实现上差异在于reduceRight是从数组的末尾开始实现。看下面这个例子：  var data = [1, 2, 3, 4];  var specialDiff = data.reduceRight(function (previous, current, index) {  if (index == 0) {  return previous + current;  }  return previous - current;  });  console.log(specialDiff); // 0  结果0是如何得到的呢？ 我们一步一步查看循环执行：  // 初始设置  index = 3, previous = initialValue = 4, current = 3  // 第一次迭代  index = 2, previous = (4- 3) = 1, current = 2  // 第二次迭代  index = 1, previous = (1 - 2) = -1, current = 1  // 第三次迭代  index = 0, previous = (-1 + 1) = 0, current = undefined (退出) | |
|  |  | |
|  | 本文为低版本IE扩展的Array方法我都合并到一个JS中了，您可以轻轻的右键这里或下载或查看：[es5-array.js](http://www.zhangxinxu.com/study/201304/es5-array.js" \t "http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/_blank)  以上所有未IE扩展的方法都是自己根据理解写的，虽然多番测试，难免还会有细节遗漏的，欢迎指出来。 | |

es6

|  |  |  |
| --- | --- | --- |
| 简述 | | forEach map filter  some every  indexOf lastIndexOf  reduce reduceRight |
| from of fill  find findIndex includes  .key() .value() .entried()  copyWithin |
|  | | var obj = {  "0": "abc",  length: 1  };  Array.from(obj); //[ 'abc' ]  Array() // []  Array(3) // [, , ,]  Array(3, 11, 8) // [3, 11, 8]  Array.of() // []  Array.of(undefined) // [undefined]  Array.of(1) // [1]  Array.of(1, 2) // [1, 2]  ['a', 'b', 'c'].fill(7) // [7, 7, 7]  [1, 4, -5, 10].find(n => n < 0) //-5  [NaN].findIndex(y => Object.is(NaN, y)) // 0  [NaN].indexOf(NaN) // -1  [1, 2, 3].includes(4) // false  [1, 2, NaN].includes(NaN) // true  for (let index of['a', 'b'].keys()) {}  for (let elem of['a', 'b'].values()) {}  for (let[index, elem]of['a', 'b'].entries()) {}  // Array.prototype.copyWithin(target, start = 0, end = this.length) |
| from | Array.from方法用于将两类对象转为真正的数组：  1、类似数组的对象（array-like object）  2、可遍历（iterable）的对象（包括ES6新增的数据结构Set和Map）。   |  |  | | --- | --- | | 类数组对象 | let arrayLike = {  '0': 'a',  '1': 'b',  '2': 'c',  length: 3  };  // ES5的写法  var arr1 = [].slice.call(arrayLike); // ['a', 'b', 'c']  // ES6的写法  let arr2 = Array.from(arrayLike); // ['a', 'b', 'c'] | | NodeList对象 | let ps = document.querySelectorAll('p');  Array.from(ps).forEach(function (p) {  console.log(p);  }); | | arguments对象 | function foo() {  var args = Array.from(arguments);  // ...  } | | string | Array.from('hello')  // ['h', 'e', 'l', 'l', 'o'] | | set | let namesSet = new Set(['a', 'b'])  Array.from(namesSet) // ['a', 'b'] | | array | 如果参数是一个真正的数组，Array.from会返回一个一模一样的新数组。  Array.from([1, 2, 3])  // [1, 2, 3] | |  |  | |  |  | |  |  | |  |  | |  |  |   值得提醒的是，扩展运算符（...）也可以将某些数据结构转为数组。  // arguments对象  function foo() {  var args = [...arguments];  }  // NodeList对象  [...document.querySelectorAll('div')]  扩展运算符背后调用的是遍历器接口（Symbol.iterator），如果一个对象没有部署这个接口，就无法转换。Array.from方法还支持类似数组的对象。所谓类似数组的对象，本质特征只有一点，即必须有length属性。因此，任何有length属性的对象，都可以通过Array.from方法转为数组，而此时扩展运算符就无法转换。  Array.from({ length: 3 });  // [ undefined, undefined, undefined ]  上面代码中，Array.from返回了一个具有三个成员的数组，每个位置的值都是undefined。扩展运算符转换不了这个对象。  对于还没有部署该方法的浏览器，可以用Array.prototype.slice方法替代。  const toArray = (() =>  Array.from ? Array.from : obj => [].slice.call(obj)  )();  Array.from还可以接受第二个参数，作用类似于数组的map方法，用来对每个元素进行处理，将处理后的值放入返回的数组。  Array.from(arrayLike, x => x \* x);  // 等同于  Array.from(arrayLike).map(x => x \* x);  Array.from([1, 2, 3], (x) => x \* x)  // [1, 4, 9]  下面的例子是取出一组DOM节点的文本内容。  let spans = document.querySelectorAll('span.name');  // map()  let names1 = Array.prototype.map.call(spans, s => s.textContent);  // Array.from()  let names2 = Array.from(spans, s => s.textContent)  下面的例子将数组中布尔值为false的成员转为0。  Array.from([1, , 2, , 3], (n) => n || 0)  // [1, 0, 2, 0, 3]  另一个例子是返回各种数据的类型。  function typesOf () {  return Array.from(arguments, value => typeof value)  }  typesOf(null, [], NaN)  // ['object', 'object', 'number']  如果map函数里面用到了this关键字，还可以传入Array.from的第三个参数，用来绑定this。  Array.from()可以将各种值转为真正的数组，并且还提供map功能。这实际上意味着，只要有一个原始的数据结构，你就可以先对它的值进行处理，然后转成规范的数组结构，进而就可以使用数量众多的数组方法。  Array.from({ length: 2 }, () => 'jack')  // ['jack', 'jack']  上面代码中，Array.from的第一个参数指定了第二个参数运行的次数。这种特性可以让该方法的用法变得非常灵活。  Array.from()的另一个应用是，将字符串转为数组，然后返回字符串的长度。因为它能正确处理各种Unicode字符，可以避免JavaScript将大于\uFFFF的Unicode字符，算作两个字符的bug。  function countSymbols(string) {  return Array.from(string).length;  } | |
| of | Array.of方法用于将一组值，转换为数组。  Array.of(3, 11, 8) // [3,11,8]  Array.of(3) // [3]  Array.of(3).length // 1  这个方法的主要目的，是弥补数组构造函数Array()的不足。因为参数个数的不同，会导致Array()的行为有差异。  Array() // []  Array(3) // [, , ,]  Array(3, 11, 8) // [3, 11, 8]  上面代码中，Array方法没有参数、一个参数、三个参数时，返回结果都不一样。只有当参数个数不少于2个时，Array()才会返回由参数组成的新数组。参数个数只有一个时，实际上是指定数组的长度。  Array.of基本上可以用来替代Array()或new Array()，并且不存在由于参数不同而导致的重载。它的行为非常统一。  Array.of() // []  Array.of(undefined) // [undefined]  Array.of(1) // [1]  Array.of(1, 2) // [1, 2]  Array.of总是返回参数值组成的数组。如果没有参数，就返回一个空数组。  Array.of方法可以用下面的代码模拟实现。  function ArrayOf(){  return [].slice.call(arguments);  } | |
| fill | 数组实例的fill()  fill方法使用给定值，填充一个数组。  ['a', 'b', 'c'].fill(7) // [7, 7, 7]  new Array(3).fill(7) // [7, 7, 7]  上面代码表明，fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去。  fill方法还可以接受第二个和第三个参数，用于指定填充的起始位置和结束位置。  ['a', 'b', 'c'].fill(7, 1, 2) // ['a', 7, 'c']  上面代码表示，fill方法从1号位开始，向原数组填充7，到2号位之前结束。 | |
| find | 数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。  [1, 4, -5, 10].find(n => n < 0)  // -5  上面代码找出数组中第一个小于0的成员。  [1, 5, 10, 15].find(val => val > 9);  // 10  上面代码中，find方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组 | |
| findIndex | 数组实例的findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。  [1, 5, 10, 15].findIndex(val => val > 9); // 2  这两个方法都可以接受第二个参数，用来绑定回调函数的this对象。  另外，这两个方法都可以发现NaN，弥补了数组的indexOf方法的不足。  [NaN].indexOf(NaN)  // -1  [NaN].findIndex(y => Object.is(NaN, y))  // 0  上面代码中，indexOf方法无法识别数组的NaN成员，但是findIndex方法可以借助Object.is方法做到。 | |
| includes | 1. 数组实例的 includes()   Array.prototype.includes方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。ES2016 引入了该方法。  [1, 2, 3].includes(2) // true  [1, 2, 3].includes(4) // false  [1, 2, NaN].includes(NaN) // true  该方法的第二个参数表示搜索的起始位置，默认为0。如果第二个参数为负数，则表示倒数的位置，如果这时它大于数组长度（比如第二个参数为-4，但数组长度为3），则会重置为从0开始。  [1, 2, 3].includes(3, 3); // false  [1, 2, 3].includes(3, -1); // true   1. 与indexOf   没有该方法之前，我们通常使用数组的indexOf方法，检查是否包含某个值。  if (arr.indexOf(el) !== -1) {  // ...  }  indexOf方法有两个缺点，  一是不够语义化，它的含义是找到参数值的第一个出现位置，所以要去比较是否不等于-1，表达起来不够直观。  二是，它内部使用严格相等运算符（===）进行判断，这会导致对NaN的误判。  [NaN].indexOf(NaN) // -1  includes使用的是不一样的判断算法，就没有这个问题。  [NaN].includes(NaN) // true  下面代码用来检查当前环境是否支持该方法，如果不支持，部署一个简易的替代版本。  const contains = (() =>  Array.prototype.includes  ? (arr, value) => arr.includes(value)  : (arr, value) => arr.some(el => el === value)  )();  contains(['foo', 'bar'], 'baz'); // => false  另外，Map 和 Set 数据结构有一个has方法，需要注意与includes区分。  Map 结构的has方法，是用来查找键名的，比如Map.prototype.has(key)、WeakMap.prototype.has(key)、Reflect.has(target, propertyKey)  Set 结构的has方法，是用来查找值的，比如Set.prototype.has(value)、WeakSet.prototype.has(value) | |
| keys values entries | 数组实例的 entries() keys() values()  ES6 提供三个新的方法——entries()，keys()和values()——用于遍历数组。它们都返回一个遍历器对象（详见《Iterator》一章），可以用for...of循环进行遍历，唯一的区别是keys()是对键名的遍历、values()是对键值的遍历，entries()是对键值对的遍历。  for (let index of ['a', 'b'].keys()) {  console.log(index);  }  // 0  // 1  for (let elem of ['a', 'b'].values()) {  console.log(elem);  }  // 'a'  // 'b'  for (let [index, elem] of ['a', 'b'].entries()) {  console.log(index, elem);  }  // 0 "a"  // 1 "b"  如果不使用for...of循环，可以手动调用遍历器对象的next方法，进行遍历。  let letter = ['a', 'b', 'c'];  let entries = letter.entries();  console.log(entries.next().value); // [0, 'a']  console.log(entries.next().value); // [1, 'b']  console.log(entries.next().value); // [2, 'c'] | |
| copyWithin | 数组实例的copyWithin方法，在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。  Array.prototype.copyWithin(target, start = 0, end = this.length) | |
| 解构 | <https://juejin.im/post/58ee092eac502e006bb94d2c>  解构的应用   1. 遍历空数组   JavaScript数组其实是天生“稀疏”的。稀疏数组其实是一个很重要的概念：  A sparse array is one in which the elements do not have contiguous indexes starting at 0.  从定义来看，稀疏数组就是没有从0开始的连续的index。  那么什么样会有稀疏数组？原因无外乎：  有没有被赋值的项；  有被删除（delete）的项  我们从下面这个例子来看：  const arr = new Array(4);//[ , , , ]  为了解决这个问题，比如，我想得到一个每一项值为其index的数组，长度为4，可以这样做：  const arr = Array.apply(null, new Array(4));  arr.map((elem, index) => index);// [0, 1, 2, 3]  当然，我们有一个更好的方法，就是使用ES6的展开符特性：  const arr = [...new Array(4)];  arr.map((elem, index) => index);// [0, 1, 2, 3]   1. 给方法传递一个空参数   如果你想调用某个方法，但是忽略这个方法的某个参数，那么正常情况下，这样做是会报错的：  function method (a1, a2, a3) { console.log('ok'); }  method('parameter1', , 'parameter3');// Uncaught SyntaxError: Unexpected token ,  在实际开发中，这样的场景其实屡见不鲜。通常的做法是，将这个函数参数传递为null或者undefined:  method('parameter1', null, 'parameter3') // or  method('parameter1', undefined, 'parameter3');  我个人其实并不喜欢用null来代替，因为在JavaScript中，null会被当作一个object来处理，这其实是很奇怪的。但是在ES6中，借助展开符和数组特性，我们能更好地实现上述做法。  上文提到JavaScript中数组其实是天生稀疏的，所以，给一个数组传递一个空值是没有问题的。因此，我们这样做：  method(...['parameter1', , 'parameter3']);// "p1",undefined,"p3"   1. 数组去重   数组去重，是一个老生常谈的话题。实现方式真的已经很多了。但是我其实一直以来不明白Array构造函数的原型上，为什么没有一个“官方”的方法，来产生一个不重复的数组或者完成数组去重的功能。ES6展开符的出现，成为了一种“官方”解决方案。  我们使用展开符，结合Set构造函数，便可以产生一个不含重复项的数组：  const arr = [...new Set([1, 2, 3, 3])]// [1, 2, 3]  其实， NaN != NaN 对数组去重的不同方法会产生不同影响。  在上述方法当中，我们会得到：  const arr = [...new Set([1, 2, 3, 3, NaN, NaN])]// [1, 2, 3, NaN] | |

[合并数组方法](https://segmentfault.com/a/1190000005169551)concat push对比

|  |  |
| --- | --- |
| concat | concat方法将创建一个新的数组，然后将调用它的对象（this指向的对象）中的元素以及所有参数中的数组类型的参数中的元素以及非数组类型的参数本身按照顺序放入这个新数组,并返回该数组。  **concat方法并不修改调用它的对象（this指向的对象）和参数中的各个数组本身的值，而是将他们的每个元素拷贝一份放在组合成的新数组中。**  原数组中的元素有两种被拷贝的方式：  1.对象引用（非对象直接量）：concat方法会复制对象引用放到组合的新数组里，原数组和新数组中的对象引用都指向同一个实际的对象，所以，当实际的对象被修改时,两个数组也同时会被修改，如：原数组和新数组里都有一个对象obj，如果改变其属性，则原数组与新数组里的obj属性都发生变化。  2.字符串和数字（是原始值，而不是包装原始值的String和Number对象）：concat方法会复制字符串和数字的值放到新数组里。 |
| 1. 连接两个数组  var a = ["a", "b", "c"];  var b = [1, 2, 3];  var ab = a.concat(b);  // ab: ["a", "b", "c", 1, 2, 3];  // 原数组 a 和 b 未被修改  2. 连接三个数组  var num1 = [1, 2, 3];  var num2 = [4, 5, 6];  var num3 = [7, 8, 9];  var nums = num1.concat(num2, num3);  console.log(nums);  // nums:[1, 2, 3, 4, 5, 6, 7, 8, 9]  // 原数组 num1, num2, num3 未被修改  var nums = [].concat(num1, num2, num3);  console.log(nums);//[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]；合并数组不影响原数组的易用方法；  3. 将非数组值合并到数组里  var alpha = ['a', 'b', 'c'];  var alphaNumeric = alpha.concat(1, [2, 3]);  // 组成新数组 ["a", "b", "c", 1, 2, 3], 原alpha数组未被修改 |
|  |  |
| push | push()方法添加一个或多个元素到源数组的末尾，并返回数组新的长度（length属性值），影响this数组  arr.push(element1, ..., elementN)  1. 添加元素到数组  var sports = ["soccer", "baseball"];  var total = sports.push("football", "swimming");  console.log(sports); // ["soccer", "baseball", "football", "swimming"]  console.log(total); // 4  2. 合并两个数组  //错误方法1  var arr1 = ['1', '2'];  var arr2 = ['3', '4'];  var total = arr1.push(arr2);  log(arr1); // [ '1', '2', [ '3', '4' ] ]  log(total);//3  //错误方法1  var arr1 = ['1', '2'];  var arr2 = ['3', '4'];  var total = Array.prototype.push.call(arr1, arr2);  log(arr1); // [ '1', '2', [ '3', '4' ] ]  log(total);//3  //应该用apply方法，将被push的数组当成参数数组。  var arr1 = ['1', '2'];  var arr2 = ['3', '4'];  var total = Array.prototype.push.apply(arr1, arr2);  log(arr1);// [ '1', '2', '3', '4' ]  log(total);//4 |
|  | Array.prototype.concat()返回一个新数组，不会改变原数组，但是会占用较多内存。  Array.prototype.push()会改变this数组，返回最新length属性，占用内存较少。 |

splice

|  |  |  |
| --- | --- | --- |
|  | 定义和用法：splice() 方法向/从数组中添加/删除项目，然后返回被删除的项目。  注释：该方法会改变原始数组。  语法：arrayObject.splice(index,howmany,item1,.....,itemX)  参数 描述  index 必需。整数，规定添加/删除项目的位置，使用负数可从数组结尾处规定位置。  howmany 必需。要删除的项目数量。如果设置为 0，则不会删除项目。  item1, ..., itemX 可选。向数组添加的新项目。 | |
| var arr=[1,2,3,4,5,6,7];  var arr2=arr.splice(1,2);  ll(arr); //[ 1, 4, 5, 6, 7 ]  ll(arr2); //[ 2, 3 ] | |
| var arr=[1,2,3,4,5,6,7];  var arr2=arr.splice(1,2,"asdf","123","asdfasdf");  ll(arr); //[ 1, 'asdf', '123', 'asdfasdf', 4, 5, 6, 7 ]  ll(arr2); //[ 2, 3 ] | |
| var arr=[1,2,3,4,5,6,7];  var arr2=arr.splice(1,2,["asdf","123","asdfasdf"]);  ll(arr); //[ 1, [ 'asdf', '123', 'asdfasdf' ], 4, 5, 6, 7 ]  ll(arr2); //[ 2, 3 ] | |
| var arr=[1,2,3,4,5,6,7];  var arr2=arr.splice(-1,1,"asdf");  ll(arr); // [ 1, 2, 3, 4, 5, 6, 'asdf' ]  ll(arr2); // [7]  //将最后一个元素置换为另外一个元素 | |
| var arr=[1,2,3,4,5,6,7];  var arr2=arr.slice(-1,2);  ll(arr); // [ 1, 2, 3, 4, 5, 6, 7 ]  ll(arr2); // [] | var arr=[1,2,3,4,5,6,7];  var arr2=arr.splice(-1,2);  ll(arr); // [ 1, 2, 3, 4, 5, 6 ]  ll(arr2); // [7] |

[在类数组对象上重用数组方法](http://blog.csdn.net/dm_vincent/article/details/41046311):

|  |  |
| --- | --- |
|  | <https://segmentfault.com/a/1190000005169551>  <http://blog.csdn.net/dm_vincent/article/details/41046311> |
| 其他 | 我们还可以将上面这些数组方法应用在其他对象上 |
|  | 1. forEach遍历DOM元素   var eleDivs = document.getElementsByTagName("div");  Array.prototype.forEach.call(eleDivs, function(div) {  console.log("该div类名是：" + (div.className || "空"));  }); |
|  | 1. 操作字符串   var str = '123,hello';  // 反转字符串  Array.prototype.reduceRight.call(str,function(a,b){  return a+b;  }); // olleh,321  // 过滤字符串，只保留小写字母  Array.prototype.filter.call('123,hello', function(a) {  return /[a-z]/.test(a);  }).join(''); // hello  // 利用 map 遍历字符串（这个例子明显举得不太好 \*\_\*）  Array.prototype.map.call(str,function(a){  return a.toUpperCase();  }); // ["1", "2", "3", ",", "H", "E", "L", "L", "O"] |
|  | 1. 使用es6中Array.from简化   var str = '123,hello';  var strN = Array.from(str).reduceRight(function (a, b) {  return a + b;  });  console.log(strN); // solleh,321 |

数组排序

|  |  |  |
| --- | --- | --- |
|  | <http://efe.baidu.com/blog/talk-about-sort-in-front-end/>  聊聊前端排序的那些事-底层  http://www.cnblogs.com/ZxrGloria/p/5316969.html  常见的几种数组排序算法JS实现  http://www.cnblogs.com/taotingkai/p/6214367.html  数组排序（选择排序和冒泡排序）  聊聊前端排序的那些事 | |
| 排序1 | var arrSimple = [1, 8, 7, 6];  arrSimple.sort();  ll(arrSimple); | [ 1, 6, 7, 8 ] |
| var arrSimple2 = [1, 8, 7, 6];  arrSimple2.sort(function (a, b) {  return b - a  });  ll(arrSimple2); | [ 8, 7, 6, 1 ] |
| var objectList = [{  name: "jack",  age: 20  }, {  name: "tony",  age: 28  }, {  name: "stone",  age: 26  }  ];  //按年龄从小到大排序  objectList.sort(function (a, b) {  return a.age - b.age  });  ll(objectList); | [ { name: 'jack', age: 20 },  { name: 'stone', age: 26 },  { name: 'tony', age: 28 } ] |
| **function keysrt**(key,desc) {  **return function**(a,b){  **return** desc ? ~~(a[key] < b[key]) : ~~(a[key] > b[key]);  } } **var ary**=[{id:9,name:"syl"},{id:2,name:"cby"}]; **ary**.sort(**keysrt**('name',**true**)); **ary**.sort(**keysrt**('name',**false**)); **ary**.sort(**keysrt**('id',**false**)); //[ { id: 9, name: 'syl' }, { id: 2, name: 'cby' } ] //[ { id: 2, name: 'cby' }, { id: 9, name: 'syl' } ] //[ { id: 2, name: 'cby' }, { id: 9, name: 'syl' } ] | 利用符号进行的类型转换,转换成数字类型 ~~true == 1 ~~false == 0 ~~"" == 0 ~~[] == 0  ~~undefined ==0 ~~!undefined == 1 ~~null == 0 ~~!null == 1 |

练习

|  |  |  |
| --- | --- | --- |
| Array.indexOf() | // Array.indexOf()函数的兼容性重写 if (!Array.prototype.indexOf) {  Array.prototype.indexOf = function(ele) {  // 获取数组长度  var len = this.length;  // 检查值为数字的第二个参数是否存在，默认值为0  var fromIndex = Number(arguments[1]) || 0;  // 当第二个参数小于0时，为倒序查找，相当于查找索引值为该索引加上数组长度后的值  if(fromIndex < 0) {  fromIndex += len;  }  // 从fromIndex起循环数组  while(fromIndex < len) {  // 检查fromIndex是否存在且对应的数组元素是否等于ele  if(fromIndex in this && this[fromIndex] === ele) {  return fromIndex;  }  fromIndex++;  }  // 当数组长度为0时返回不存在的信号：-1  if (len === 0) {  return -1;  }  } } | |
| 判断一个对象是否为一个数组 | <http://www.nowamagic.net/librarys/veda/detail/1250>  typeof arr object  arr instanceof Arrary true  Object.prototype.toString.call(arr) "[object Arrary]". | |
| for…in 遍历数组的隐藏问题 | for…in用来遍历一个对象中的成员（属性，方法），如果用来遍历数组的到的结果并不是预期中数组每项的值，方法也会被遍历出来  事实上很多时候我们都会给数组加上其他属性。比如 jQuery 对象就是一个数组对象加上一些扩展方法；再比如 String.prototype.match 方法返回值就是一个数组（正则表达式及其子表达式的匹配项）加上 index 和 input 两个属性。  Array.**prototype**.contains = **function** (item) {  **for** (**var i** = 0; **i** <= **this**.**length** - 1; **i**++) {  **if** (**this**[**i**] == item)**return this**[**i**];  } } **var arr** = [**"A"**, **"B"**]; **for** (**var i** = 0; **i** <= **arr**.**length** - 1; **i**++) {  log(**arr**[**i**]); } // for...in Enumeration: the method "contains" are enumerated, too **for** (**var singleStaff in arr**) {  log(**singleStaff**); } | |
| JS数组元素删除:delete splice | **var myArray**=[**"a"**,**"b"**,**"c"**]; **delete myArray**[0]; **for**(**var i**=0,**j**=**myArray**.**length**;**i**<**j**;**i**++){  **console**.log(**myArray**[**i**]);  /\* undefined b c \*/ }  **var myArray2**=[**"a"**,**"b"**,**"c"**]; **myArray2**.splice(0,1); **for**(**var i**=0,**j**=**myArray2**.**length**;**i**<**j**;**i**++){  **console**.log(**myArray2**[**i**]);  /\* b c \*/ }  delete是将元素设置为undefined，splice则是将元素排除掉了 |
| 判断一个对象是否为数组 | **function** isArray(obj) {  **return** Object.prototype.toString.call(obj) == **"[object Array]"**; } **function** \_\_getClass(object) {  **return** Object.prototype.toString.call(object).match(/^\[object\s(.\*)\]$/)[1]; };  **function** isArray(object) {  **return** object != **null** && **typeof** object === **"object"** && **'splice' in** object && **'join' in** object; } //关于iframe中的数组类型判断 **var** iframe = document.createElement(**'iframe'**); document.body.appendChild(iframe); xArray = window.frames[window.frames.length - 1].Array; **var** arr = **new** xArray(1, 2, 3); // [1,2,3] arr **instanceof** Array; // false arr.constructor === Array; // false  不能用instanceof 和 constructor来判断，原因参考：  http://perfectionkills.com/instanceof-considered-harmful-or-how-to-write-a-robust-isarray/ |
| 数组中随机获取成员 | **var** items = [12, 548 , **'a'** , 2 , 5478 , **'foo'** , 8852, , **'Doe'** , 2145 , 119]; **var**  randomItem = items[Math.floor(Math.random() \* items.length)]; |
|  |  |
|  |  |

jquery工具函数-each map...

|  |  |
| --- | --- |
|  | F:\HTML\practice\jquery\tool of jquery.html |
|  | <**meta** charset="utf-8"/> <**script** type="text/javascript" src="jquery-1.8.3.js"></**script**> <**script**>  **var log** = **function** () {  **if** (arguments.length == 2) {  console.log(arguments[0]);  console.log(":");  console.log(arguments[1]);  }  **if** (arguments.length == 1) {  console.log(arguments[0])  }  }; </**script**> |
| each | log("-----------------------each [遍历]"); **/\*  \* 通用例遍方法，不同于例遍 jQuery 对象的 $("").each() 方法，此方法可用于例遍任何对象。  \* 与\_.each(arr,function(n,i){}) arr.forEach(function(n,i,arr){})中function的参数顺序有所不同  \* P1:对象的成员或数组的索引  \* P2:索引的相应内容  \* 通过回调函数返回false，退出each循环，其它返回值将被忽略。  \* \*/  /\*遍历一个数组\*/** $.each([0, 1, 2], **function** (i, n) {  log("[each 1] " + i + " : " + n); }); **/\*遍历一个对象\*/** $.each({name: "John", lang: "JS"}, **function** (i, n) {  log("[each 2] " + i + " : " + n); }); **/\*遍历一个对象，通过if(--){return false|true;}来决定是否结束遍历过程\*/** $.each({name: "John", lang: "JS"}, **function** (i, n) {  log("[each 3] " + i + " : " + n);  **if** (n == "John") {  **return false**;  } }); |
| map  grep | log("-----------------------map [重组]"); $.map([0, 1, 2], **function** (n) {  **return** n + 4; }); // [4, 5, 6]  $.map([0, 1, 2], **function** (n) {  **return** n > 0 ? n + 1 : **null**; }); // [2, 3]  $.map([0, 1, 2], **function** (n) {  **return** [n, n + 1]; }); // [0, 1, 1, 2, 2, 3] |
| log("-----------------------grep [筛选]"); **/\*  \* P1:被筛选的对象  \* P2:筛选方式  \* P3:默认false，设置为true：取反  \* \*/ var oo1614** = $.grep([0, 1, 2], **function** (n, i) {  **return** n; }); log("oo1614", **oo1614**);//[1, 2]  **var oo1614** = $.grep([0, 1, 2], **function** (n, i) {  **return** n > 0; }); log("oo1614", **oo1614**);//[1, 2]  **var oo1617** = $.grep([0, 1, 2], **function** (n, i) {  **return** n > 0; }, **true**); log("oo1617", **oo1617**);//[0] |
| log("-----------------------map > grep"); **/\* \* 筛选数组中满足要求的元素，返回一个新数组 \* map:获得一个新数组，通过 return newValue|null; \* grep:获得一个新数组，通过 return true|false; \* map主要用来数组中元素的"平行"变化，当返回一个array null时，可以实现"纵向"变换 \* \*/  /\*map与grep使用的对比：map;\*/ var oo1622** = '1,2,3,4,5,jQuery,CSS,5'.split(','); **oo1622** = $.map(**oo1622**, **function** (num, index) {  **return** isNaN(num) ? num : **null**; }); log("oo1622", **oo1622**);//["jQuery", "CSS"]  **/\*map与grep使用的对比：grep;\*/ var oo1622** = '1,2,3,4,5,jQuery,CSS,5'.split(','); **oo1622** = $.grep(**oo1622**, **function** (num, index) {  **return** isNaN(num); }); log("oo1622", **oo1622**);//["jQuery", "CSS"] |
| extend | log("-----------------------extend 重载原型"); **/\*  \* 后面相应key's value覆盖相应的值  \* 参数：target,[object1],[objectN]  \*  \* target 本身会发生变化与返回值一致  \* object1 objectN 本身不变  \* \*/ var target** = {value: "aaa"}; **var obj1** = {validate: **false**, limit: 5, name: "foo"}; **var obj2** = {validate: **true**, name: "bar"}; log($.extend(**target**, **obj1**, **obj2**));  //{value: "aaa", validate: true, limit: 5, name: "bar"} log("target", **target**); //{value: "aaa", validate: true, limit: 5, name: "bar"} log("object1", **obj1**); //{validate: false, limit: 5, name: "foo"} log("object2", **obj2**); //{validate: true, name: "bar"} |
| **/\*  \* 是否深度拷贝  \* [deep],target,object1,[objectN]  \* P1：boolean：代表是否进行深度拷贝：[true:深度拷贝][false:浅度拷贝]  \* 默认浅拷贝  \* \*/** log($.extend(  **false**,  {},  {name: "John", location: {city: "Boston", county: "USA"}},  {last: "Resig", location: {state: "MA", county: "China"}} )); // {name:"John",last:"Resig",location:{state:"MA",county:"China"}} log($.extend(  **true**,  {},  {name: "John", location: {city: "Boston", county: "USA"}},  {last: "Resig", location: {state: "MA", county: "China"}} )); // {name:"John",last:"Resig",location:{city:"Boston",state:"MA",county:"China"}} |
| merge  unique | log("-----------------------merge unique [合并，去重]"); console.log($.merge([0, 1, 2], [2, 3, 4]));//[0, 1, 2, 2, 3, 4] console.log($.unique($.merge([0, 1, 2], [2, 3, 4])));//[4, 3, 2, 1, 0] console.log($.unique([1,"1",2,"2","2",{a:"a"},{a:"a"},{a:"a1"},[1,2],[1,2],[1,2,3]])); //[Object, Array[3], 1, Array[2], Array[2], Object, Object, "1", "2", 2] //只会处理简单的数字、字符串；即判断的基础是"===" |
| 判断 | //-----------------------inArray [index of array]s **/\*  \* 某个元素是否存在于一个数组之中：  \* 参数： P1:值 P2:数组 P3:从第几个开始  \* 返回值：index | -1  \* \*/ var arr** = [4, "Pete", 8, "John"]; $.inArray("John", **arr**); //3 $.inArray(4, **arr**); //0 $.inArray("David", **arr**); //-1 $.inArray("Pete", **arr**, 2); //-1  //-----------------------isArray $.isArray([]);  //-----------------------isNumber $.isNumeric("-10"); // true $.isNumeric(16); // true $.isNumeric(0xFF); // true $.isNumeric("0xFF"); // true $.isNumeric("8e5"); // true (exponential notation string) $.isNumeric(3.1415); // true $.isNumeric(+10); // true $.isNumeric(0144); // true (octal integer literal) $.isNumeric(""); // false $.isNumeric({}); // false (empty object) $.isNumeric(NaN); // false $.isNumeric(**null**); // false $.isNumeric(**true**); // false $.isNumeric(Infinity); // false $.isNumeric(undefined); // false //-----------------------isFunciton **/\* 非函数： 浏览器提供的函数，比如'log'； DOM元素操作方法，比如 'getAttribute' ； \*/** $.isFunction(**new** Date()); //false $.isFunction(Math.min()); //false //-----------------------type $.type(undefined); // "undefined" $.type(); // "undefined" $.type(window.undefined); // "undefined" $.type(**null**); // "null"  $.type(3); // "number" $.type("test"); // "string" $.type(**true**); // "boolean" $.type(**new** Date()); // "date" $.type(/test/); // "regexp"  $.type([]); // "array" $.type({name: "syl"}); //"object"  $.type(**function** () { }); // "function"  $.type(**new** Error()); // "error" as of jQuery 1.9  //其他一切都将返回"object" |
|  | [$.isPlainObject()](http://api.jquery.com/jQuery.isPlainObject/" \t "http://www.ruanyifeng.com/blog/2011/07/_blank) 判断某个参数是否为用"{}"或"new Object"建立的对象。  [$.makeArray()](http://api.jquery.com/jQuery.makeArray/" \t "http://www.ruanyifeng.com/blog/2011/07/_blank) 将对象转化为数组。  [$.trim()](http://api.jquery.com/jQuery.trim/" \t "http://www.ruanyifeng.com/blog/2011/07/_blank) 去除字符串两端的空格。 |

其他

|  |  |
| --- | --- |
| for...of循环 | ES6 借鉴 C++、Java、C# 和 Python 语言，引入了for...of循环，作为遍历所有数据结构的统一的方法。  一个数据结构只要部署了Symbol.iterator属性，就被视为具有iterator接口，就可以用for...of循环遍历它的成员。也就是说，for...of循环内部调用的是数据结构的Symbol.iterator方法。  for...of循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如arguments对象、DOM NodeList 对象）、后文的 Generator 对象，以及字符串。 |
|  | 数组原生具备iterator接口（即默认部署了Symbol.iterator属性），for...of循环本质上就是调用这个接口产生的遍历器，可以用下面的代码证明。  const arr = ['red', 'green', 'blue'];  for (let v of arr) {  console.log(v); // red green blue  }  const obj = {};  obj[Symbol.iterator] = arr[Symbol.iterator].bind(arr);  for (let v of obj) {  console.log(v); // red green blue  }  上面代码中，空对象obj部署了数组arr的Symbol.iterator属性，结果obj的for...of循环，产生了与arr完全一样的结果。  for ...in循环读取键名，for...of循环读取键值。如果要通过for...of循环，获取数组的索引，可以借助数组实例的entries方法和keys方法，参见《数组的扩展》章节。  for ...of循环调用遍历器接口，数组的遍历器接口只返回具有数字索引的属性。这一点跟for...in循环也不一样。  Set 和 Map 结构也原生具有 Iterator 接口，可以直接使用for...of循环。  ES6的数组、Set、Map 都部署了entries keys values ，调用后都返回遍历器对象；for (let pair of arr.entries()) {  for...of循环用于字符串、DOM NodeList 对象、arguments对象类数组对象；并不是所有类似数组的对象都具有 Iterator 接口，一个简便的解决方法，就是使用Array.from方法将其转为数组。 |
|  |  |

## Math

|  |  |
| --- | --- |
|  | 方法 描述 |
|  | [acos(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_acos.asp) 返回数的反余弦值。  [asin(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_asin.asp) 返回数的反正弦值。  [atan(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_atan.asp) 以介于 -PI/2 与 PI/2 弧度之间的数值来返回 x 的反正切值。  [atan2(y,x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_atan2.asp) 返回从 x 轴到点 (x,y) 的角度（介于 -PI/2 与 PI/2 弧度之间）。  [cos(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_cos.asp) 返回数的余弦。  [exp(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_exp.asp) 返回 e 的指数。  [log(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_log.asp) 返回数的自然对数（底为e）。  [pow(x,y)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_pow.asp) 返回 x 的 y 次幂。  [sin(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_sin.asp) 返回数的正弦。  [sqrt(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_sqrt.asp) 返回数的平方根。  [tan(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_tan.asp) 返回角的正切。  [toSource()](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_tosource_math.asp) 返回该对象的源代码。  [valueOf()](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_valueof_math.asp) 返回 Math 对象的原始值。    [abs(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_abs.asp) 返回数的绝对值。  **[max(x,y)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_max.asp) 返回 x 和 y 中的最高值。**  **[min(x,y)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_min.asp) 返回 x 和 y 中的最低值。**  [random()](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_random.asp) 返回 0 ~ 1 之间的随机数。(0,1)  [round(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_round.asp) 把数四舍五入为最接近的整数。  [ceil(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_ceil.asp) 对数进行上舍入。(向正方向)  [floor(x)](mk:@MSITStore:F:\\HTML\\W3CSchool%20%20b.chm::/www.w3school.com.cn/jsref/jsref_floor.asp) 对数进行下舍入。 |
|  | Math.round(Math.random()\*10)：[0,10]  Math.round(Math.random()):0101010101010110  Math.ceil(Math.random()\*10):(0,10] 向上  Math.ceil(Math.random()):1111111111111111  Math.ceil(Math.random()\*10):[0,10) 向下  Math.floor(Math.random()):000000000000000 |
|  | **var minH** = **Math**.*min*.apply(**null**, **pin**);  一个数组中的最小值 |

## Date

|  |  |
| --- | --- |
| new Date() | new Date()  //Date {Fri Aug 02 2013 16:50:33 GMT+0800 (China Standard Time)} new Date(milliseconds)  //Date {Fri Aug 02 2013 16:53:26 GMT+0800 (China Standard Time)} new Date("2013/08/02")  //Date {Fri Aug 02 2013 00:00:00 GMT+0800 (China Standard Time)} new Date(year,month,day,hours,minutes,seconds,ms)  前三种方式没有什么问题，但第四种得到的结果回合我们预期的不一致  new Date(2013,08,02)  //Date {Mon Sep 02 2013 00:00:00 GMT+0800 (China Standard Time)}  我们可以看到，传入的月份是08，返回的结果却是九月。这是因为Date对象的月份是从0开始计数的（天却不是），即0代表一月，1代表二月…11代表12月。在调用Date实例的getMonth方法时尤其要注意  var d = new Date(2012, 4, 15); // 2012年5月15日  d.getMonth(); // 结果为4 |
|  | new Date() 可以接受:  没有参数就返回当前时间  一个参数 x 就返回当前时间1970年1月1日， + x 毫秒,Unix的人知道为什么  new Date(1，1，1) 返回1901年，2月1日.因为你知道，第一个1意思是1900年后的1年，第二个1是这一年的第二个月（因此二月） —— 二月的正确索引在索引表中为 1 ，第三个 1 显然是这个月的第一天，所以是 1 —— 因为有时索引确实从1 开始。  new Date(2016年，1，1) 不会从2016年追加到1900年。它仅仅表示2016年。 |
| Date.parse() | Date.parse方法可以识别两种格式的字符串参数（标准的长日期格式，比如带星期的那种，也可以识别，不过不常用）：  1. "M/d/yyyy": 美国的日期显示格式。如果年传入2位则作为 19xx 处理  2."yyyy-MM-dd" 或 "yyyy/MM/dd": 注意月和日都必须是两位  Date.parse 的返回结果不是一个Date对象，而是从1970-01-01午夜（GMT）到给定日期之间的毫秒数。可以用Date的构造函数将其转换为Date对象。  new Date(Date.parse("8/2/2012"));  // 正确识别为2012年8月2日  new Date(Date.parse("2012-08-02"));  // 正确识别为2012年8月2日  new Date(Date.parse("2012-8-2"));  // 不能识别 |
| tostring | **var date**=**new** Date(); **log**(**date**.toString()); **log**(**date**.toDateString()); **log**(**date**.toLocaleString()); //Wed Dec 21 2016 22:24:48 GMT+0800 (中国标准时间) //Wed Dec 21 2016 //2016-12-21 22:24:48 |
| getYear  getFullYear | 使用getYear()函数的本意是获取年份，以2010年为例，在IE中是可以正确获取年份：2010，但是在FF等浏览器下则为：110。  原因：  在 FF等浏览器内 getYear 返回的是 "当前年份-1900" 的值（即年份基数是1900）  而 IE则是 当today的年份大于等于2000的时，直接将1900加上了，返回的 2010。    使用getFullYear()在IE和FF中都可以正确获取年份：2010  结论：使用js来获取年份都使用：getFullYear() |

其他

|  |  |  |
| --- | --- | --- |
| 一行代码 | | 今天我想分享Addy Osmani的[一行代码](https://gist.github.com/addyosmani/fd3999ea7fce242756b1" \t "http://ourjs.com/detail/_blank) ，这行代码对于你调试你的CSS是很有用的。为了可读性，我把它变成了3行。  注\* Addy Osmani 是Google Chrome开发工程师，他前几天开发的[字符串解析模板](http://ourjs.com/detail/548e3f388a34fa3204000008" \t "http://ourjs.com/detail/_blank), 马上被兼容[最新ES6标准的io.js](http://ourjs.com/detail/54b887fd232227083e00000e" \t "http://ourjs.com/detail/_blank)采纳。  [].forEach.call($$("\*"),function(a){   a.style.outline="1px solid #"+(~~(Math.random()\*(1<<24))).toString(16) })  尝试在你浏览器的Console(F12)中运行一下，你会发现页面被不同的颜色块高亮了，这个方法非常简单，但是你自己写的话可能产生非常多的代码，让我们来研究一下它是怎么实现的。  选择一个页面上的所有元素  我们首先需要选择页面上的所有元素。Addy使用了只能在console调试工具中使用的$$函数，你可以在console中输入$$('a')自己试一下。它会返回当前页面的所有anchor（链接）元素。  $$与document.querySelectorAll是等价的。所以$$('\*')与 document.querySelectorAll('\*')是等价的，有兴趣的话可以看看[$$和$选择器的历史](http://ourjs.com/detail/54ab768a5695544119000007" \t "http://ourjs.com/detail/_blank)。   对于我来说，$$的使用已经让我学到了很多。但是选择页面上的所有元素的知识还有很多。Mathias Bynens就在评论中指出document.all 其实也能选取选用元素，而且兼容所有主流浏览器。  遍历所有的元素  现在我们有了一个所有元素的节点列表（NodeList）,现在我们想遍历它们，并给他们加上有颜色的边框。我们先看看能从这行代码里发现什么：  [].forEach.call( $$('\*'), function( element ) { /\* And the modification code here \*/ });  NodeList看起来像一个Array（数组），你可以使用中括号来访问他们的节点，而且你还可以通过length属性知道它有多少元素。但是它并没有实现Array的所有接口，因此使用 $$('\*').forEach 会返回错误，在JavaScript的世界里，有一堆看起来像Array但其实不是的对象。如function中的arguments对象。因此在他们身上通过call和apply来应用数组的方法是非常有用的。我之前写过一篇文章来解析它们的用法，下面是一个例子  function say(name) {  console.log( this + ' ' + name ); }   say.call( 'hola', 'Mike' ); // 打印输出 'hola Mike'  之前的一行代码使用 [].forEach.call 代替 Array.prototype.forEach.call 减少了代码的编写量 ( 另外一个很有意思的地方 )；如果$$('\*')返回是个数组的话，它与$$('\*').forEach是等价的。  如果你看看评论，还有人使用for(i=0;A=$$('\*');)让代码变得更短，但是它在全局对象中注入了变量。  你可以带上var声明，如  for(var i=0,B=document.querySelectorAll('\*');A=B[i++];){ /\* your code here \*/ }  其中i和B将只声明在console的上下文中。  改变元素的颜色  让元素有一个漂亮的边框，这行代码使用了CSS的outline属性。有一点你可能不知道，在CSS渲染的盒子模型（Box Model）中，outline并不会改变元素及其布局的位置。因此这比使用border属性要好得多，所以这一部分其实并不难理解  a.style.outline="1px solid #" + color  怎样定义颜色值其实是比较有意思的  ~~(Math.random()\*(1<<24))).toString(16)  我不是特别懂位运算，因此我最喜欢这一段。  我们想构造的其实是一个16进制的颜色值，像白色FFFFFF，蓝色0000FF等等。  首先我们学到了可以使用数字类型的toString方法进行十进制到16进制的转换。  其实你可以用它进行任意进制的转换  (30).toString();   // "30" (30).toString(10); // "30" (30).toString(16); // "1e" 16进制 (30).toString(2); // "11110" 二进制 (30).toString(36); // "u" 36 是最大允许的进制  因此16进制中的ffffff其实是 parseInt("ffffff", 16) == 16777215，16777215是2^24 - 1的值  因此左位移操作乖以一个随机数 Math.random()\*(1<<24) 可以得到一个0 到 16777216之间的值  但是还不够，Math.random返回的是一个浮点数字，我们只需要整数部，这里使用了"~"操作符（按位取反操作）。  这行代码并不关心正负值。因此通过两次取返就可以得到纯整数部，我们还可以将~~视为parseInt的简写：  var a = 12.34, // ~~a = 12     b = -1231.8754, // ~~b = -1231     c = 3213.000001 // ~~c = 3213 ;   ~~a == parseInt(a, 10); // true ~~b == parseInt(b, 10); // true ~~c == parseInt(c, 10); // true  如果你仔细看评论你会知道使用 按位或 "|"操作符也可以得到相同的结果。  ~~a == 0|a == parseInt(a, 10) ~~b == 0|b == parseInt(b, 10) ~~c == 0|c == parseInt(c, 10)  我们最终得到了一个 0 到 16777216之间的随机数，然后使用toString(16)转换成16进制，它就是这样工作的。 |
| 常用的jsLib | | 我后台里面用这些。 <script type="text/javascript" src="inc/javascript/jquery.js"></script> <script type="text/javascript" src="inc/javascript/jquery.form.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ext.js"></script> <script type="text/javascript" src="inc/javascript/common.js"></script> <script type="text/javascript" src="inc/javascript/private.js"></script> <script type="text/javascript" src="inc/javascript/xh\_config.js"></script> <link rel="stylesheet" href="inc/css/common.css" type="text/css" />  <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.core.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.draggable.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.resizable.js"></script> <script type="text/javascript" src="inc/javascript/jquery.autocomplete.pack.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.dialog.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/external/bgiframe/jquery.bgiframe.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.datepicker.js"></script> <script type="text/javascript" src="inc/javascript/jquery\_ui/ui/ui.datetimepicker.js"></script> <script type="text/javascript" src="inc/javascript/ajaxfileupload.js"></script> <script type="text/javascript" src="inc/javascript/jquery.blockUI.js"></script> |
| 一行代码分析 | | 关于if (obj.length === +obj.length)判断  underscore.js 用来操作数组数据十分的方便，今天在阅读underscore.js源码的时候发现了一个判断语句，一时没看懂是什么意思，后来仔细分析了一下，也到网上查阅相关的资料。  if (obj.length === +obj.length){  for (var i = 0,length = obj.length; i < length; i++){  if (iterator.call(context, obj[i], i, obj) === breaker)  return;  }  }  源码如上，请注意if判断,看上去很奇怪很难懂，先把这句话的意思表达一起，如果换一种写法你肯定就会明白了，其实这句话等价于另外一种写法 :  if (typeof obj.length === "number" && !isNaN(obj.length))  看上去这样写似乎更能够让人看懂，而且也能达到效果，但为什么作者要这样写呢，接下来我们需要关注两个地方：  第一个就是‘===’，全等符号  第二个就是‘+’ 加号  先说一下全等号的作用，在一般的if判断中，我们更多的是使用‘==’，这种模式比如if(2 == ‘2’)，这时候我们得到的返回结果为true，但在‘===’，这种模式下，我们得到的结果是false。这是因为全等符号是不会将比较的对象进行类型转换的。  再说一下‘+’，在此处说所起到的作用,你可以动手做一个实验，在控制台中执行下面这句话+’2’,你可以看到返回值为数字类型的2，其实不难理解，’+’号其实是将后面跟的操作数转型成了数字类型。  我们在是想一下作者这样写的目的，如果obj是一个string类型，如"abc",我们可以拿到length属性，如果是一个function，或者一个数组，我们都可以拿到他们的length属性，但如果是一个object类型的数据，它可能是不包含length属性的。对于非数组、非字符串、非函数类型的数据，我们可以尝试使用for in循环来遍历数据。这样看来其实作者这样的写法，更是想能区分数组类型的数据或者类数组数据如字符串等。 |
| 谷歌调试 | | 调试快捷键  F5 ： 开始调试  Shift+F5: 停止调试  F10 ： 调试到下一句，这里是单步跟踪  F11 ： 调试到下一句，跟进函数内部  Shift+F11: 从当前函数中跳出  Ctrl+F10: 调试到光标所在位置  F9 ： 设置 [ 取消 ] 断点  Alt +F9: 高级断点设置 跟踪调试 1.尽量使用快捷键时行调试 2.观察调试信息 3.高级中断设置 异常调试 重试 － > 取消 － > 调试 函数堆栈，用 variables 或者 call stack 窗口 Release 调试  1.经常测试你的 Debug 和 Release 版本 2.不要移除调试代码，如用 ASSERT, TRACE 等 。 3.初始化变量，特别是全局变量， malloc 的内存， **new** 的内存 4.当你移除某个资源时，确保你移除了所有跟这个资源相关的申明[主要是在 resouce.**h** 文中 ] 5.使用 3 或者 4 级的警告级编译你的代码，并确保没有警告， project->setting->**c**/**c**++->warninglevel(中文版是项目 - > 属性 - > C/C++-> 常规 － > 警告等级 ) 6.\_debug 改成 NDEBUG 进行调试， project->setting->C/C++->Preprocessordefinitions [ 中 文版是项目 - > 属性 - > C/C++-> 预处理器 - > 预处理定义] [这里是 debug 和 Release 编译的重要不同之一] 7.在 Release 中调试源代码， project->setting->C/C++->debug info 选择programDataBase [中文版是项目 - >属性->C/C++->常规->调试信息格式->用于"编辑并继续"的程序数据库]， project－>setting->link 选上Generate debug info [中文版是项目->属性->链接器->调试->生成调试信息] 8.走读代码，特别关注堆栈和指针 |
| json | [JSON](http://caibaojian.com/t/json" \o "json)([javascript](http://caibaojian.com/t/javascript" \o "javascript常用代码大全) Object Notation) 是一种轻量级的数据交换格式，采用完全独立于语言的文本格式，是理想的数据交换格式。同时，JSON是 JavaScript 原生格式，这意味着在 JavaScript 中处理 JSON数据不须要任何特殊的 API 或工具包。  在JSON中，有两种结构：对象和数组。  1.对象  一个对象以"{"开始，"}"结束。每个"key"后跟一":"，"‘key/value’ 对"之间运用 ","分隔。  packJson = {"name":"caibaojian.com", "password":"111"}  2.数组  packJson = [{"name":"caibaojian.com", "password":"111"}, {"name":"tony", "password":"111"}];  数组是值的有序集合。一个数组以"["开始，"]"结束。值之间运用 ","分隔。 | |
| 在数据传输流程中，[json](http://caibaojian.com/t/json" \o "View all posts in json" \t "http://caibaojian.com/_blank)是以文本，即字符串的形式传递的，而[JS](http://caibaojian.com/t/javascript" \o "JS)操作的是JSON对象，所以，JSON对象和JSON字符串之间的相互转换是关键。例如：  JSON字符串：var [json](http://caibaojian.com/t/json" \o "View all posts in json" \t "http://caibaojian.com/_blank)Str = '{"name":"caibaojian", "password":"1111"}';  JSON对象：var jsonObj = {"name":"caibaojian.com", "password":"1111"};  1.String转换为Json对象：eval('(' + jsonStr + ')');  2.Json对象转换为String字符串：jsonObj.toJSONString(); | |