**原生JS**

前注：

这里不提简单的常见函数，那些内容请看我的网易云课堂的js的word文件。这里的目的，是深挖原生JS。

**（一）BOM功能：**

浏览器对象模型

①弹出新浏览器窗口的功能；window.open

②移动、缩放和关闭浏览器窗口的功能；

③提供浏览器详细信息的navigator对象；

④提供浏览器所加载页面的详细信息的location对象；

⑤提供用户显示器分辨率详细信息的screen对象；

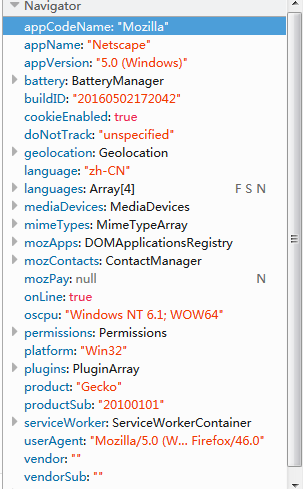
⑥对cookies的支持；

⑦像XMLHttpRequest和IE的ActiveXOjbect这样的自定义对象。

**（二）navigator对象**

1. *Navigator {vendorSub: "", productSub: "20030107", vendor: "Google Inc.", maxTouchPoints: 0, hardwareConcurrency: 4…}*
   1. appCodeName:"Mozilla"
   2. appName:"Netscape"
   3. appVersion:"5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36"
   4. cookieEnabled:true
   5. credentials:CredentialsContainer
   6. doNotTrack:null
   7. geolocation:Geolocation
   8. hardwareConcurrency:4
   9. language:"zh-CN"
   10. languages:Array[2]
   11. maxTouchPoints:0
   12. mediaDevices:MediaDevices
   13. mimeTypes:MimeTypeArray
   14. onLine:true
   15. permissions:Permissions
   16. platform:"Win32"
   17. plugins:PluginArray
   18. presentation:Presentation
   19. product:"Gecko"
   20. productSub:"20030107"
   21. serviceWorker:ServiceWorkerContainer
   22. userAgent:"Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/52.0.2743.82 Safari/537.36"
   23. vendor:"Google Inc."
   24. vendorSub:""
   25. webkitPersistentStorage:DeprecatedStorageQuota
   26. webkitTemporaryStorage:DeprecatedStorageQuota
   27. \_\_proto\_\_:Navigator

以上是chrome的，



以上是火狐的，

属性解释：

先上代码：

**$**(document).ready(function () {  
 **$**("#a").click(function () {  
 **txt** = "<p>浏览器代号: " + navigator.appCodeName + "</p>";  
 **txt** += "<p>浏览器名称: " + navigator.**appName** + "</p>";  
 **txt** += "<p>浏览器版本: " + navigator.**appVersion** + "</p>";  
 **txt** += "<p>启用Cookies: " + navigator.cookieEnabled + "</p>";  
 **txt** += "<p>硬件平台: " + navigator.**platform** + "</p>";  
 **txt** += "<p>用户代理: " + navigator.**userAgent** + "</p>";  
 **txt** += "<p>用户代理语言: " + navigator.systemLanguage + "</p>";  
 document.getElementById("a").innerHTML = **txt**;  
 })  
})

结果：

360浏览器：

浏览器代号: Mozilla

浏览器名称: Netscape

浏览器版本: 5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36

启用Cookies: true

硬件平台: Win32

用户代理: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.101 Safari/537.36

用户代理语言: undefined

chrome：

浏览器代号: Mozilla

浏览器名称: Netscape

浏览器版本: 5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36

启用Cookies: true

硬件平台: Win32

用户代理: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/51.0.2704.103 Safari/537.36

用户代理语言: undefined

safari：

浏览器代号: Mozilla  
  
浏览器名称: Netscape  
  
浏览器版本: 5.0 (iPhone; CPU iPhone OS 9\_3\_3 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13G34 Safari/601.1  
  
启用Cookies: true  
  
硬件平台: iPhone  
  
用户代理: Mozilla/5.0 (iPhone; CPU iPhone OS 9\_3\_3 like Mac OS X) AppleWebKit/601.1.46 (KHTML, like Gecko) Version/9.0 Mobile/13G34 Safari/601.1  
  
用户代理语言: undefined

①appCodeName：只读字符串，用于声明浏览器的代码名，

经测试，在chrome，firefox，safari里的值都是Mozilla；

手机端的uc浏览器测试失败

②appName基本是一样的；

③appVersion返回浏览器的平台和版本信息；

④cookieEnabled表示cookie是否能用；

⑤platform返回运行浏览器的操作系统平台；

⑥userAgent返回由客户机发送服务器的 user-agent 头部的值。

⑦systemLanguage返回 OS 使用的默认语言。

⑧online：是否在线；

**（三）location对象**

属性列表（chrome）

假如是这样的url：<https://www.baidu.com/?key1=a&key2=b#abc>

其属性为：

1. ancestorOrigins:DOMStringList
2. assign:*()*
3. hash:"#abc"
4. host:"www.baidu.com"
5. hostname:"www.baidu.com"
6. href:"https://www.baidu.com/?key1=a&key2=b#abc"
7. origin:"https://www.baidu.com"
8. pathname:"/"
9. port:""
10. protocol:"https:"
11. reload:*reload()*
12. replace:*()*
13. search:"?key1=a&key2=b"
14. toString:*toString()*
15. valueOf:*valueOf()*
16. \_\_proto\_\_:Location

①hash表示其哈希值，即url之中，#之后的内容，包括#；

②host表示根url，即如果是abc.com/d/e/f，它的值只包含abc.com；

他包含端口（假如端口为80则不显示），但不包含http开头的部分

③hostname同样表示根url，但不包含端口，假如端口不是80，他依然是abc.com，但不包含http开头的部分

④href是当前页面的完整路径；

⑤origin：同样是url，但包含http或者https开头的部分，也包括之后的端口。（注：不兼容IE）

⑥pathname：指的是根目录之下的路径，其具体值为从host之后部分，到问号之前的部分；例如abc.com/d/e/f，其pathname值为/d/e/f

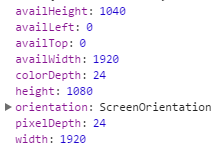
⑦protocol返回url中的协议部分，即http或者https；

⑧search指问号之后，#之前的部分；包括？不包括#；在get请求中发送中常用，用于保存不重要的内容；

**（四）screen对象**

这是跟屏幕相关的属性对象；

chrome下，属性如下：（不包含通过prototype继承的属性）



①availHeight：屏幕可用高度（除windows任务栏），因此会比实际分辨率小一些；

②availWidth：屏幕可用宽度，一般都和实际分辨率保持相同。这两条属性，用qq自带的截图工具，可以轻易获取到这个宽高。

③availLeft和availTop：一般情况下，都是0。经过实测，即使调整分辨率，让屏幕有黑边（即左右没有实际填充满显示屏），也不影响这个值。

④height、width：指屏幕的实际高度和宽度，可以知道屏幕的分辨率如何。

⑤colorDepth：返回目标设备或缓冲器上的调色板的比特深度。（不明白）

⑥pixelDepth：返回屏幕的颜色分辨率（比特每像素），以上两个一般一样，大概就是指颜色比如16位之类的？IE应该不支持。

**（五）script标签**

【1】标签在创建的时候可以赋予六（七）个属性（都是可选）；

①src：最重要的属性，表示链接外部的js代码文件，如果引用外部js文件的话就需要添加这个属性。附带效果，添加这个属性后的script标签内部的代码将无效。

②async：表示应该立即下载脚本，但不应妨碍页面中的其他操作，只适用于外部脚本（加载的js文件）

示例：

<**script** src="test.js" async="async"></**script**>

说明：

（1）异步加载，举例：

<**script** src="test.js" async="async"></**script**>  
<**script**>  
 alert("2");  
</**script**>

会先执行alert(“2”)，再提示test.js的alert(“1”)

如果不加这个属性的话，会先1后2

③defer：其效果是在页面加载完后再加载执行js文件，但仅限于对外部文件，对内部的js代码是无效的。

④charset：通过此属性来指定代码的字符集，不过现在默认都是utf8，而且大多数浏览器会忽略，基本不用。

⑤type：目前默认都是text/javascript。表示编写代码的脚本语言的内容类型。

⑥language：废弃

这是W3SCHOOL的说明：

表格：

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | HTML4 | HTML5 | 备注 |
| type | 需要 | 可选 | 虽然HTML5可以不加，但个人觉得为了兼容性还是加上这个属性比较好 |
| async | 无 | 新增 | 外部文件专用 |
| xml:space | 有 | 不支持 | 因此不用 |
| charset | 有 | 有 | 非特殊情况不加 |
| src | 外部文件需要 | 外部文件需要 | 最重要的属性 |
| defer | 有 | 有 | 外部文件专用，加载完后执行js文件 |

【2】有意思的问题：

1. 假如通过document.createElement(“script”)来动态创建一个script标签在head部分或者body部分，然后将js代码通过innerHTML的方法写入其中，那么，这段js代码会不会执行

答：不会被执行

1. 以下代码会发生什么情况：

<script>  
        console.log("</script>");  
</script>

答：输出

”)

然后报错。解决办法：使用<\/script> 作为替代

【3】在html文档里，不要用<script xxxxx />这种写法，原因是在IE或者其他浏览器里很可能不兼容，是XHTML的写法。

【4】script标签，可以加载来源于外部域的js文件。

举个例子，我本地html文件，可以加载来源于百度或者其他地方的某个js文件。但需要添加http://

潜在问题是，对方必须可信，否则你可能遇见恶意代码。

【5】标签的加载改善：

1. <script></script>标签的位置，一般放于</body>标签的前方，可以减少空白页面出现的时间，从而让用户感觉页面加载速度提升了。
2. 或者添加defer=”defer”属性。假如有两个这样的标签A和B，且A前B后。我们**期望顺序**是：DOMContentLoaded事件——》A标签js代码——》B标签js代码。但我查的几个资料，说法有所区别。

* 在JavaScript高级程序设计（第三版）中，提到假如有多个延迟脚本，那么延迟脚本的顺序不一定按期望顺序执行，因此推荐只用一个延迟脚本。
* 但在<http://ued.ctrip.com/blog/script-defer-and-async.html>，这个文章中，他提到async属性的脚本会在加载完后立刻执行，顺序不确定，而延迟脚本的顺序是确定的。
* 个人倾向于后者是对的，因为在书上的小结里，又提到“延迟脚本总是按照指定他们的顺序执行”。

1. 多个JS文件之间有前后依赖关系的情况下，不要使用async属性。
2. 只在外部文件中使用defer，在内部文件不要使用，在IE6情况特殊，IE7、8会延迟在底部，Firefox中无效（估计chrome也无效）
3. 异步脚本一定会在页面的load事件之前执行，但可能在DOMContentLoaded事件之前或之后执行。

【6】noscript标签

当浏览器不支持javascript脚本，或者是浏览器支持脚本但禁用了脚本功能时，我们需要做一些特殊的处理，比如说告诉用户你需要启用javascript脚本之类。

具体办法是，使用<noscript></noscript>标签，这个元素可以包含能够出现在文档<body>中的所有元素（除script标签）。

我用百度首页来举例，在百度首页中，其内容如下：

<**noscript**>  
 <**meta** http-equiv=refresh content="0; url=http://www.baidu.com/baidu.html?from=noscript">  
</**noscript**>

他会跳转到另外一个页面，而这个页面是没有js代码的。

具体原理我不太明白，但是效果是：刷新页面，然后跳转到url=xxxx的页面下。

**（六）文档模式**

①不同文档模式对应了不同的排版引擎，不同的JS引擎，决定如何解释你的dom树和js代码

②文档模式有四种：

混杂模式（quirks mode）：

标准模式（standards mode）：

准标准模式（almost standards mode）：

超级标准模式：IE8引入。

③HTML5的文档模式只需要使用<!DOCTYPE html>即可

原因在于html5不基于SGML（维基百科https://zh.wikipedia.org/wiki/SGML)，而HTML4.01基于SGML，所以需要对DTD进行引用，才能告知浏览器文档所使用的模型。（就是标签里那一长串链接）。

④什么都不写的话，默认是混杂模式，其结果在不同浏览器上可能是有所不同的。

一般来说还是要用标准模式，保证在各个浏览器的效果是一致的。

标准模式的开启：

<!-- HTML 4.01 严格型 -->   
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "[http://www.w3.org/TR/html4/strict.dtd](http://www.w3.org/TR/html4/strict.dtd" \t "http://blog.csdn.net/cuew1987/article/details/_blank)">   
       
<!-- XHTML 1.0 严格型 -->   
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "[http://www.w3.org/TR/xhtml1/DTD/xhtml1](http://www.w3.org/TR/xhtml1/DTD/xhtml1" \t "http://blog.csdn.net/cuew1987/article/details/_blank)-

strict.dtd">

但我看百度、腾讯和google他们都是这么写的：

<!DOCTYPE html>

所以虽然不太确认这种写法在旧版本IE浏览器中能否执行，但我个人认为是可以执行的。

参考链接：

[文档模式和浏览器模式](http://blog.csdn.net/cuew1987/article/details/12754087)

http://blog.csdn.net/cuew1987/article/details/12754087

文档模式的介绍和需要考虑的事情

<http://www.cnblogs.com/fsjohnhuang/p/3817418.html>

  所以作为普通开发者的我们只需做三件事：

  1. 以有效的doctype作为文档的第一行，保证渲染模式为标准模式；

  2. 开发前设定网站的最佳运行环境范围，就是需适配的IE版本号，是否适配Webkit等；

  3. 开发并使用各种hacks为兼容性付出不懈的努力。

**（七）严格模式**

①ECMAScript5 引入了严格模式（strict mode）的概念。

严格模式是为JavaScript定义了一种不同的解析与执行模型，在这种模式下，ECMA3Script中的一些不确定的行为将得到处理，对不安全的操作也会抛出错误。

②启用方法：

整个脚本：

顶部添加 “use strict”

某个函数：

函数内部的第一行（花括号之后的第一行）添加 “use strict”

③兼容严格模式的浏览器：IE10+、FireFox 4+、Safari 5.1+、 Opcra 12+和Chrome

④在严格模式下，很多原本可以正常跑的代码，都不可以了，会抛出异常，或者结果有所区别。典型的就是：

function test(value) {  
 "use strict"  
 value = 7;  
 console.log(value, arguments[0]);  
}  
function test2(value) {  
 value = 7;  
 console.log(value, arguments[0]);  
}  
test(10);  
test2(10);

严格模式的结果是7，10；

非严格模式的结果是7，7；

原因在于，非严格模式对参数的操纵会影响arguments中的值，而严格模式是不影响的。

参考链接：

**[Javascript的严格模式](http://www.xiabingbao.com/javascript/2015/02/14/javascript-strict" \t "http://www.xiabingbao.com/javascript/2015/02/14/_self)**

<http://www.xiabingbao.com/javascript/2015/02/14/javascript-strict>

**（八）变量的声明**

应避免以下这种声明方式：

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

原因在于，这种情况下，变量a的作用域是代码块，脱离函数后被成功销毁；

但变量b变成了全局变量，在console.log(b)中可以正确的显示其结果。

但以下这种声明方式是可以的：

function test() {  
 var a = 1,  
 b = 1;  
}

**（九）变量，类型**

①**typeof**

注：输出结果都是字符串，因此可以在if语句里用===某个字符串来匹配是否是某个类型

|  |  |
| --- | --- |
| typeof以下类型变量 | 输出结果 |
| 声明未赋值／未声明变量 | undefined |
| 布尔值 | boolean |
| 字符串 | string |
| 数值 | number |
| 对象或者null | object |
| 函数 | function |

**②null和undefined**

null == undefined 返回值为true；

null === undefined 返回值为false

原因在于，undefined是null派生而来的。

**③boolean**

true == 1 返回值true

true == ‘1’ 返回值true

true === 1 返回值false

false同理

Boolean('')和Boolean(' ')的值，分别是false和true

Boolean是一个布尔类型的强制转换函数

转换后为false 的：（1）空字符串；（2）数字0；（3）null；（4）NaN；（5）undefined；（6）false本身

放在if等判断语句里，会自动转换；

**④number**

（1）八进制：赋值时，0开头，且符合要求（例如每位数字都在0~7之间），如果数字超出，忽略开头的0，当做10进制解释（不使用数字除外，会出错）；

八进制严格模式下(“use strict”）无效

十六进制：赋值时，0x开头（或0X），如果使用非范围内数字字母，会出错，字母可大写可小写；

十进制：其他正常赋值时。

计算时，会转为十进制来计算。

（2）浮点数：即小数；

* 存在精度问题（17位）；
* 末尾0很多时（var a = 0.000000000000000000001或var a = 10000000000000000000000）会被转化为1e然后后面跟一个整数或者负数；
* 不要计算两个浮点数相加后和另外一个浮点数他们之间的结果是否相等（0.1+0.2 == 0.3的结果是false，其值为0.30000000000000004），除此之外还有0.1+0.7结果是0.7999999999999999等
* 最小值大概是5e-324，最大值是1.7xxxx e+308。假如某次计算超出这个数值，会转换为Infinity（正无穷）或 -Infinity（负无穷）；Infinity减去或除以Infinity的结果是NaN，乘以或者加上依然是Infinity；Infinity - 1结果还是Infinity；Infinity\*0结果是NaN，/0结果是Infinity；1/0结果也是Infinity（火狐和chrome，其他未验证），而不是书上说的NaN，但0/0是结果是NaN

（3）NaN 非数值，是一种特殊的数值

0/0结果是NaN，其他数值除以0则不是；

NaN == NaN 的结果是false，包括三个等号的结果

isNaN()可以用于测试一个数值是否是NaN，能被转化为数值的为false（比如字符串），不能转化为数值的为true，比如字符串为true，true/false为false，undefined为true等；

可以检测对象，但并非必然是true。检测方式是调用对象的valueOf()方法，之后是toString()，根据其值决定是true还是false（有一个为false即为false，都是true才是true），；空对象返回true，但例如以下情况则返回false：

var a = {}; //true

var a = { valueOf: 1}; //true

var a = { valueOf: function(){return 1}}; //false

var a = { toString: function(){return 1;}} //false

var a = { toString: function(){return "a";}} //true

（4）数值转换

三个转换函数：Number(), parseInt(), parseFloat()

Number()

|  |  |
| --- | --- |
| 源 | 目标 |
| true/false | 1或0 |
| 数字 | 返回数字 |
| null | 0 |
| undefined | NaN |
| 字符串：纯数字 | 作为十进制数字转换，忽略开头0，因此不存在八进制识别 |
| 字符串：浮点数（无其他干扰） | 浮点数 |
| 字符串：十六进制格式 | 十六进制数识别，然后转换为十进制 |
| 字符串：空 | 0 |
| 字符串其他：例如在以上情况下有其他字符干扰 | NaN |

parseInt()

|  |  |
| --- | --- |
| 源 | 结果 |
| 字符串开头有非0~9非空格字符 | NaN |
| 空格开头的字符串 | 忽略空格后看上一行，不符合的话看下一行 |
| 0x开头后面跟数字比如0x1 | 当十六进制数字识别，剩下的看第六行 |
| ~~0开头后面跟数字比如011~~  失效 | ~~当八进制数字识别，剩下看第六行，~~  实际测试失效，例如parseInt("070")  返回70而不是56 |
| 普通数字开头 | 当做十进制数字识别 |
| 数字后跟字母或空格  （重要） | 如果是十六进制数，遇见非a~f字母后，从该位置开始忽略后面的；  如果是十进制数字，遇见字母后，从第一个字母开始忽略后面的 |
| 小数 | 识别为整数 |

当parseInt()当存在第二个参数时，第二个参数用于作为转换时的基制，即多少进制；假如以下是

例如parseInt("11",2)的结果是3，parseInt("11",11)的结果是12。前者是作为二进制识别，后者是作为十一进制识别。

第二个参数为1时NaN，0则使用默认，当第一个参数不是十进制数字时，返回值为0，例如parseInt("0x11",2)作为二进制解析时，其值为0（x不能作为二进制识别）；

0x开头的，第二个参数只能使用16（转换为十六进制），否则无法识别（正常识别，识别结果为0）。

当使用第二个参数时，则不会将其设法转换为十六进制，而是作为普通字符串来处理。

**按照教程所说：**

**如果我们要默认其以某个进制来转换，应该显式的在第二个参数声明他，例如第二个参数写10，默认其为10进制，这样的话，当其看起来像16进制数字，可以避免用十六进制来解释**

parseFloat()

和parseInt()类似的识别方式，只不过他会忽视开头的0或者空格，识别到非浮点数为止。也就是说，他不认十六进制或者其他。

例如：parseFloat("0x1.1")返回0（不识别十六进制）；

parseFloat("0000.11")返回0.11（前面加若干空格结果也一样）

parseFloat(" 0 000.11")返回0，因为从开始识别的位置之后，他不会忽视空白字符。

**⑤string类型**

【1】转义字符：

|  |  |
| --- | --- |
| 字符 | 表示 |
| \n | 换行 |
| \t | 制表符 |
| \b | 空格 |
| \r | 回车 |
| \f | 走纸换页  （然而并不明白所以给个搜索链接https://www.google.com/#newwindow=1&q=%E8%B5%B0%E7%BA%B8%E6%8D%A2%E9%A1%B5） |
| \xnn | nn是十六进制字符，x需要小写。需要2位n。  用于表示字符，其是ASCII字符集，比如41转换为十进制为65，ASCII中65表示A |
| \unnnn | 类似上面，x小写，n十六进制，四位n都要有，表示Unicode字符集。 |
| 对象无toString方法 | "[object Object]" |
| 对象有toString方法 | toString方法的返回值 |

【2】.length属性

字符串都有；

不能正确识别汉字是两个字符的长度；

转义字符是转义后的长度。例如”\x41”表示字母A，他的长度则为1

【3】其他类型转为字符串：

（1）调用toString()方法；

* .toString()方法在加上参数后，可以将一个数字转换为2~36进制中的任何一个；例如：(113).toString(30)的返回值是"3n"
* true转为”true”；
* 普通数字转为数字；
* 十六进制数字转为10进制数字后输出。例如：(0x1a).toString()返回26
* null和undefined没有这个方法，会出错。
* 字符串本身也有这个方法。但这个方法不能被覆写。例如var a=”a”;a.toString = function(){return "b"}; 这个时候再调用a.toString()，则返回值依然为”a”，而不是”b”

（2）使用String()方法，被转换的变量作为参数

假如一个方法有toString()方法，则调用之；

假如没有（只剩null和undefined），则null变为”null”，undefined变为”undefined”

（3）通过+号连接符，隐式调用toString方法

比如说拼接；

var a={};

a.toString = function(){return "bb"}

a+”aa”的返回值是”bbaa”

⑥object对象

【1】每个object对象都具有的方法：

（1）constructor：显示一个对象构造函数的方法；

用例一：查看某个对象的构造函数。

假如有一个我们自定义的函数

function test(a, b) {  
 this.**a** = a;  
 this.**b** = b;  
**}**

我们new一个他的实例

var m = new test();

这个时候m的constructor是什么呢，就是上面那个函数；

用例二：

我们如何判断一个对象是否是某个类的实例，很简单，就像在上面那样，调用constructor属性和原类进行比值（可以用三个等号）。

例如：

function test(a, b) {  
 this.**a** = a;  
 this.**b** = b;  
}  
var m = new test();  
console.log(m.**constructor** === test);

其返回子是true

类似，数组的constructor的值是Array（没有引号），字符串是String，数字是Number，布尔值是Boolean

var test = new Array;  
console.log(test.constructor === Array);

【2】hasOwnPrototype(“某属性”)

（1）用于检测某个对象的实例是否具有某个属性；

（2）只检测当前实例有没有，跟原型有没有无关；

（3）原型链上继承来的属性无法检测到。

示例：

function test(a) {  
 this.**a** = a;  
}  
var m = new test(1);  
console.log(m.hasOwnProperty("a")); //检测有没有a属性，原型有，所以有  
console.log(m.hasOwnProperty("b")); //检测有没有b属性，原型没有，目前没有  
m.b = function(){};  
console.log(m.hasOwnProperty("b")); //检测有没有b属性，原型没有，但后来赋值了，所以有

返回值依次是：true, false, true

证明了（1）和（2）

function test(a) {  
 this.**a** = 1;  
}  
var a = new test();  
test.prototype.test2 = function () {  
 return "1";  
}  
console.log(a.test2());  
console.log(a.hasOwnProperty("test2"));

返回值依次为1和false

说明实例的原型继承到的方法，是被实例所继承的，但是无法通过hasOwnPrototype来检测。

**扩展：**

之前的情况，可以通过in来检测，例如；

console.log("test2" in a);

返回值为true

for in方法也是可以检测到的

for (var i in a) {  
 console.log(i)  
}

输出：a和test2

但据说老版本的浏览器上是不可以的，参照《for in的缺陷》：

<http://www.cnblogs.com/snandy/archive/2011/03/03/1970132.html>

【3】obj1.prototype.isPrototypeOf(obj2)

（1）用于检测调用这个方法的对象obj1，是否是参数obj2的原型；

例如：

function test(a) {  
 this.**a** = 1;  
}  
var b = new test();  
console.log(test.prototype.isPrototypeOf(b));

返回值为true；

说明test是b的原型。

**（2）\*\*\*\*\*\*\*\*\*\*\*\*\* 问题 \*\*\*\*\*\*\*\*\*\*\*\*\*：**

按照说明，他会检查原型链上的继承，但我自己测试后，假如是继承，那么返回值依然为false，如下：

function test() {  
}  
function test2() {  
}  
test2.prototype.test = test;  
var m = new test2();  
console.log(test2.prototype.isPrototypeOf(m));  
console.log(test.prototype.isPrototypeOf(m));

返回值依次为true和false

test的确在test2的原型链上，但是返回值为false。搞不懂。

也许是我搞错了原型链上的继承。

【4】.propertyIsEnumerable(属性名)

（1）检测某个属性能否使用for in语句来枚举，参数需要是字符串类型

（2）无视原型链上的内容（即无法检测原型链上的）

例如：

function test() { //原型  
 this.**a** = 1;  
**}**test.prototype.**b** = "test"; //test2方法是test的一个属性  
var m = new test();  
console.log(m.propertyIsEnumerable("b"));  
console.log(m.propertyIsEnumerable("a"));

var n = {**a**: 1, **b**: 2};  
console.log(n.propertyIsEnumerable("a"));

返回值依次为false，true，true

在这里，m是test的实例；b是test继承的一个属性；

因此，这两个都可以通过for in来显示，但由于a是test的属性，而b是通过继承来的，因此propertyIsEnumberable是无法检测到b的。故一个返回false，一个返回true

n有属性a，因此其返回值为true

【5】toLocaleString()

（1）返回对象的字符串表示；

（2）该字符串与执行环境的地区对应。

具体不太清楚，先放置吧，也许和下面的有关系？

【6】toString()

（1）返回对象的字符串表示。可以通过手动覆写的方法，使其可以用于对象可以和字符串相加，然后返回字符串。例如：

var m = {**a**: 1, **b**: 2};  
m.toString = function () {  
 return "m";  
}  
console.log(m + "n");

返回值是字符串mn

此时：

console.log(m.toLocaleString());

输出的值也是字符串m

【7】valueOf()

（1）按照说明：返回对象的字符串、数值、或布尔值表示。通常与toString()的返回值相同。但并不是这样，个人觉得是和对象本身的值是一样的

例如：

var m = {**a**: 1, **b**: 2};  
console.log(m.toString());  
console.log(m.valueOf());

输出值是：字符串[object Object]和对象Object {a: 1, b: 2}

即使覆写m的toString()方法，valueOf()输出的值依然是Object {a: 1, b: 2}

**（十）一元操作符**

①++和-- （右边是两个减号）

【1】不能直接对某个字符串使用运算，例如”1”++ 是会报错的

【2】但可以将字符串赋给变量后，对变量使用，例如：

var m = "1";  
console.log(++m);

输出2，类型是number

【3】对纯数字可以相加，对非纯数字（字母或字母加数字），则返回NaN

【4】false视为0，true视为1；

【5】对象如果没有valueOf方法，则返回NaN；如果有valueOf方法，则其返回值表示这个对象的值。（Number()方法的返回值和调用valueOf方法的返回值是相同的）

【6】++和--是一样的；

②+和 -

【1】对数字没有影响，只表示正负，或转换正负；

【2】对其他类型，会尝试像使用Number()那样对其进行转换