

Javascript 学习笔记

基础语法

概述

Js 是基于原型的面向对象—prototype-based 00

Java 等时基于类的面向对象—class-based 00

页面方法

```
1.alert()  
2.document.write()  
3.console.log();
```

Onclick 属性

可以向 HTML 标签的 onclick 属性中添加 js 语句

```
<button onclick="alert('讨厌！你点人家干嘛！')">点我一下</button>
```

Js 语句写在 href 中，点击超链接时也会执行 js 代码：

```
<a href="javascript:alert('点一下')">hit me</a>
```

这种形式会造成结构与行为耦合，不建议使用

外部引入 JS

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

注意: `script` 标签一旦用于外部引入, 就不能在其中再添加其他 `js` 语句

需要则另起一 `script` 标签

代码注意事项

`Script` 标签不能出现在 `js` 内容中, 浏览器会识别并结束

Label 语句

需要联合 `continue/break` 语句, 应用于嵌套循环, 一般的 `break/continue` 只能退出一层循环, 采用 `label` 时一次即可退出所有循环。

语法:

LabelName:

For 循环

Break/continue labelName;

Eg:

```
var num = 0;
start:
for(var i =0;i<10;i++){
    for(var j = 0;j<10;j++){
        if (i==5&&j==5) {
            break start;//一次即可退出所有循环
        }
        num++;
    }
}
//num = 55;
```

With 语句

作用: 简化多次编写同一个对象的工作

格式: `with(expression){ statement}`

Eg:

With (a) {

Var q = search; // q = a.search;

```
    Var h = hostname; //h = a.hostname;  
}
```

原理:

语句中的代码块中，每个变量首先被认为是一个局部变量，在局部变量中找不到其定义，则会查询该对象是否有同名的属性，有则赋值。

注意:

With 会导致性能下降，同时造成调试困难，大型应用不建议使用。

Switch 语句

注意:

在 js 中，switch(agru) 的参数没有限制，可以是字符串、对象、数字等甚至是条件<>=等布尔类型

script 标签有属性

1.type
2.src
3.defer
defer="defer"

延迟作用

告诉浏览器立即下载外部 js 文件，但是延迟执行

4.async 异步脚本

同 defer

但是它不保证按几个外部文件的顺序执行

window.open ()

`window.open([URL], [窗口名称], [参数字符串])`

参数:

URL: 可选参数, 在窗口中要显示网页的网址或路径。如果省略这个参数, 或者它的值是空字符串, 那么窗口就不显示任何文档。

窗口名称: 可选参数, 被打开窗口的名称。

1. 该名称由字母、数字和下划线字符组成。
2. "_top"、"_blank"、"_self"具有特殊意义的名称。

`_blank`: 在新窗口显示目标网页

`_self`: 在当前窗口显示目标网页

`_top`: 框架网页中在上部窗口中显示目标网页

3. 相同 name 的窗口只能创建一个, 要想创建多个窗口则 name 不能相同。
4. name 不能包含有空格。

参数字符串: 可选参数, 设置窗口参数, 各参数用逗号隔开。

参数表:

top=
left=
width=
height=

menubar =yes/no 菜单栏

toolbar=yes/no 工具栏

scrollbars=yes/no 滚动条

status=yes/no 状态栏

例子:

```
window.open("http://www.baidu.com","_blank","width =10,
height=20,toolbar = yes")
```

数据类型

String
Number
Boolean
Null
Undefined
Object

数据类型转换

将其他的类型转化为 String Number Boolean

-to String:

方法 1: 调用 `toString()`;

注意: Null Undefined 无 `toString()`

方法 2: 调用 `String(value)`;

对于 Number String Boolean 实际上就是调用 `toString()`,而 undefined

null 则是直接转换

-to Number:

调用 `Number()`;

转换不了则返回 NaN

True → 1

False → 0

特殊:

对于字符串转数字:

`parseInt()`
`parseFloat()`

将字符串中有效的数字取出来,但是只返回最前面的一个数字,第一个数字必须是 0-9;

-to Boolean

`Var a = Boolean(数字):` 0 与 NaN 是 false, 其余是 true

`Var a = Boolean(字符串):` 只有空串""为 false

`Var a = Boolean(null):` false

`Var a = Boolean(undefined):` false

一个对象实例: true

逻辑运算

















































































































`&&` `||` `!`

非布尔值: 先转换为布尔值

Unicode 编码

引用格式: `\uxxxx` (十六进制)

在 HTML 网页中: `&#n` (n:十进制数);

| | 260 | 261 | 262 | 263 | 264 | 265 | 266 | 267 | 268 | 269 | 26A | 26B | 26C | 26D | 26E | 26F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

Eg: `☠` 骷髅头

Noscript 标签

`noscript` 标签是一种防御性措施，如果你的浏览器不支持 JS 或者是设置过高的安全级别，就会显示相应的提示信息

```
<noscript>你的浏览器不支持JS</noscript>
<script>noscript与script是并列的，并非包含关系</script>
```

NO, `noscript` 也可以嵌入 `script` 中

Delete

删除, `delete expression`(一个对象的属性)

对于所有情况都是 `true`，除非属性是一个自己不可配置的属性，在这种情况下，非严格模式返回 `false`。

注意：与通常的看法不同，`delete` 操作符与直接释放内存无关。内存管理 通过断开引用来间接完成的

`delete` 操作符会从某个对象上移除指定属性。成功删除的时候回返回 `true`，否则返回 `false`。但是，以下情况需要重点考虑：

- 如果你试图删除的属性不存在，那么 `delete` 将不会起任何作用，但仍会返回 `true`
- 如果对象的原型链上有一个与待删除属性同名的属性，那么删除属性之后，对象会使用原型链上的那个属性（也就是说，`delete` 操作只会在自身的属性上起作用）
- 任何使用 `var` 声明的属性不能从全局作用域或函数的作用域中删除。
 - 这样的话，`delete` 操作不能删除任何在全局作用域中的函数（无论这个函数是来自于函数声明或函数表达式）
 - 除了在全局作用域中的函数不能被删除，在对象(object)中的函数是能够用 `delete` 操作删除的。
- 任何用 `let` 或 `const` 声明的属性不能够从它被声明的作用域中删除。
- 不可设置的(Non-configurable)属性不能被移除。这意味着像 `Math`，`Array`，`Object` 内置对象的属性以及使用 `Object.defineProperty()` 方法设置为不可设置的属性不能被删除。

Onfocus

`Onfocus` 属性表示标签获取焦点时进行的操作

集合篇

哈希表

Hash table: 根据 key-value 访问存储数据的集合

```
Var ht = {'1':1,'2':2}
```

集合操作和闭包

```
var f = function (list,op) {  
    for (var i=0;i<list.length;i++){  
        list[i] = op(list[i]);  
    }  
};  
var a =Array(1,2,3);  
f(a,function (a) {  
    return a+1;  
});  
document.writeln(a);
```

实现 ArrayList

字符串篇

模式匹配

Str.match(regex)

返回值:

不匹配：第一个不匹配的字符

匹配：null

```
var s =String("12a34k");  
document.writeln(s.match(/\D/));//a  
s = String('12345');  
document.writeln(s.match(/\D/));//null
```

Str.search(reg/str)

搜索到返回下标，否则返回-1

Str.toLowerCase()

Str.toUpperCase()

Str.fromCharCode();

函数篇

函数参数的理解

javascript 的参数实际上是由内置数组 arguments 决定的

每当传进来时给 arguments 赋值，数组大小是参数的个数

- 1.当传入参数不足时。用 undefined 补足
- 2.当传入参数过多时，也给其赋值，只是一般应用是不会用到。

Js 都是按值传递，不按引用传递

按值传递与按引用传递

- 1.对于一般的变量，是按值传递的，传递后两个变量互不影响

- 2.对于对象实例而言，是按引用传递的，变量间相互影响
- 3.函数的参数都是按值传递，实参传入，其值不受影响，即使传入对象实例也是按值传递。不对，传入对象是按引用传递的。
- 4.参数是数组是，是按引用传递的，会改变值

函数内部属性

两个特殊对象：

- 1.arguments—类对象数组，包含传入函数的所有参数

a.callee 属性

该属性是一个指针，指向拥有这个 arguments 对象的函数

```
function ff(n) {  
    if (n<=1){  
        return 1;  
    }  
    else {  
        return n*arguments.callee(n-1);  
    }  
}
```

- 2.对象 this—引用的是函数数据以执行的环境

函数的属性和方法

每个函数包含两个属性：length、prototype

- 1.length:函数的参数个数

FuncName.length

- 2.prototype—保存了一个实例的所有方法

不可枚举的，不能用于 for-in 语句

函数内置方法

funcName.func();

1.apply(作用域, arguments/Array 对象);—只接受 2 个参数

作用：在特定的作用域调用函数，即设置函数体内 this 的值

```
function sum(a,b) {  
    return a+b;  
}  
function callSum(a,b) {  
    return sum.apply(this,arguments);  
    //or:  
    return sum.apply(this,[a,b]);  
}
```

2.call(作用域, 参数 1, 参数 2...)

作用于 apply 相同，但是除作用域外参数是分别传递的。

```
function callSum(a,b) {  
    return sum.call(this,a,b);  
}
```

注意：

call(), apply() 的更大的功能是扩充作用域

其好处是：对象不需要和方法有任何耦合关系

```
sayColor.call(this);           //red  
sayColor.call(window);         //red  
sayColor.call(o);              //blue
```

3.bind()—创建一个函数实例，将其 this 被绑定到传给 bind() 的函数

```
window.color = "red";  
var o = { color: "blue" };  
  
function sayColor(){  
    alert(this.color);  
}  
var objectSayColor = sayColor.bind(o);  
objectSayColor();    //blue
```

基本包装类型

Boolean Number String

使用包装：包装、拆箱

检测包装类型： instanceof

```
var obj = new Object("some text");  
alert(obj instanceof String);    //true
```

String 及字符类型

字符方法：

- 1.charAt()
- 2.charCodeAt()

字符串操作方法：

- 1.concat()
- 2.slice()
- 3.substr()
- 3.substring()
- 4.indexOf()
- 5.lastIndexOf()
- 6.trim()

字符串模式匹配方法

- 1.text. match(正则 / RegExp 对象)

```
var text = 'cat,bat,sat,fat';  
var pat = /\.fat/;  
var marches = text.match(pat); // 匹配第一个
```

2. text . search (正则 / RegExp 对象)

返回匹配项第一次出现的下标

3. text.replace(被替换串(仅替换一个)/正则, 替换串/函数)

4.

未完待续: P127

5. str1.localeCompare(str2);

按字典序

在前: 返回 -1

相同: 返回 0

在后: 返回 1

一般是区分大小写的

6. String.fromCharCode(a,b...) — 静态方法

接收一个或多个字符编码值, 然后返回对应字符串

数值处理

1. toFixed(n) n: 小数位数

2. toExponential(n) — e 表示法, n: 小数位数

```
var a = 10000;  
var b = a.toExponential(2); // 1.00e+4
```

3. toPrecision(n)

```
var num = 99;  
alert(num.toPrecision(1));    //"1e+2"  
alert(num.toPrecision(2));    //"99"  
alert(num.toPrecision(3));    //"99.0"
```

4.toString()

```
var num = 10;  
alert(num.toString());        //"10"  
alert(num.toString(2));       //"1010"  
alert(num.toString(8));       //"12"  
alert(num.toString(10));      //"10"  
alert(num.toString(16));      //"a"
```

5.toFixed(n)—Number 类型按照参数指定的小数位数返回字符串

```
var num = 10;  
alert(num.toFixed(2));        //"10.00"
```

对象篇

对象类型

1. 内建对象

在 ECMA 标准中定义的对象，如 Math String Function

2. 宿主对象

由 JS 运行环境提供的对象，如 BOM DOM

2. 自定义对象

创建对象

有两种方式：

```
1. var obj = new Object();  
   obj.name = "xutao";  
   obj.engname = "Edwin";  
   obj.age =21;
```

此种方式不能用数字作为属性名

除了使用点运算符，还可以使用中括号：

```
Obj["name"] = 123;
```

2.对象字面量表示法

```
Var person={  
  "Name" : "xutao",  
  "Age": 21  
}
```

对象字面量向函数传递参数

```
function f(args){  
  document.write(args.name+": "+args.age);  
}  
f({name:'xutao',age:21});
```

通过 f(

```
{  
Key:value,  
}
```

)向函数传递大量参数

内存管理

解除对象：将其赋值为 null

```
Eg: var a = new Object();  
    a = null;
```


单体内置对象

Global 对象

Global 对象十分特别，不属于其他任何对象的属性和方法，最终都是它的属性和方法，如 `isNaN()`，`isFinite()`，`parseInt()`，`parseFloat()`

其他方法：

1.URL 编码方法(URI:Uniform Resource Identifiers 通用资源标识符)

-`encodeURIComponent()`—空格被替换成%20，其余不变

```
show(encodeURIComponent('http://www.baidu.com/ s d'));
```

-`encodeURIComponentComponent()`—替换所有非字母、数字

2.`eval(paracter)`—ECMA 最强大的方法

相当于一个完整的 ECMAScript 解析器

参数：唯一参数—要执行的 js 代码的字符串形式

3. Global 对象的属性

Global 对象还包含一些属性，其中一部分属性已经在本书前面介绍过了。例如，特殊的值 `undefined`、`NaN` 以及 `Infinity` 都是 Global 对象的属性。此外，所有原生引用类型的构造函数，像 `Object` 和 `Function`，也都是 Global 对象的属性。下表列出了 Global 对象的所有属性。

| 属 性 | 说 明 | 属 性 | 说 明 |
|------------------------|----------------------------|-----------------------------|----------------------------------|
| <code>undefined</code> | 特殊值 <code>undefined</code> | <code>Date</code> | 构造函数 <code>Date</code> |
| <code>NaN</code> | 特殊值 <code>NaN</code> | <code>RegExp</code> | 构造函数 <code>RegExp</code> |
| <code>Infinity</code> | 特殊值 <code>Infinity</code> | <code>Error</code> | 构造函数 <code>Error</code> |
| <code>Object</code> | 构造函数 <code>Object</code> | <code>EvalError</code> | 构造函数 <code>EvalError</code> |
| <code>Array</code> | 构造函数 <code>Array</code> | <code>RangeError</code> | 构造函数 <code>RangeError</code> |
| <code>Function</code> | 构造函数 <code>Function</code> | <code>ReferenceError</code> | 构造函数 <code>ReferenceError</code> |
| <code>Boolean</code> | 构造函数 <code>Boolean</code> | <code>SyntaxError</code> | 构造函数 <code>SyntaxError</code> |
| <code>String</code> | 构造函数 <code>String</code> | <code>TypeError</code> | 构造函数 <code>TypeError</code> |
| <code>Number</code> | 构造函数 <code>Number</code> | <code>URIError</code> | 构造函数 <code>URIError</code> |

ECMAScript 5 明确禁止给 `undefined`、`NaN` 和 `Infinity` 赋值，这样做即使在非严格模式下也会导致错误。

Window 对象

在全局作用域中声明的所有变量和函数，都成为了 `window` 的对象的属性。

```

var color = 'red';
function sayColor() {
    alert(window.color);
    //全局变量中所有的属性都是window的属性
}
window.sayColor();

```

Math 对象

对象:

| 属 性 | 说 明 |
|--------------|---------------------|
| Math.E | 自然对数的底数, 即常量e的值 |
| Math.LN10 | 10的自然对数 |
| Math.LN2 | 2的自然对数 |
| Math.LOG2E | 以2为底e的对数 |
| Math.LOG10E | 以10为底e的对数 |
| Math.PI | π 的值 |
| Math.SQRT1_2 | 1/2的平方根 (即2的平方根的倒数) |
| Math.SQRT2 | 2的平方根 |

Math.max() Math.min()

Math.ceil() Math.floor() Math.round()----舍入

Math.random(): 0—1

| 方 法 | 说 明 | 方 法 | 说 明 |
|----------------------|----------------|------------------|------------|
| Math.abs(num) | 返回num的绝对值 | Math.asin(x) | 返回x的反正弦值 |
| Math.exp(num) | 返回Math.E的num次幂 | Math.atan(x) | 返回x的反正切值 |
| Math.log(num) | 返回num的自然对数 | Math.atan2(y, x) | 返回y/x的反正切值 |
| Math.pow(num, power) | 返回num的power次幂 | Math.cos(x) | 返回x的余弦值 |
| Math.sqrt(num) | 返回num的平方根 | Math.sin(x) | 返回x的正弦值 |
| Math.acos(x) | 返回x的反余弦值 | Math.tan(x) | 返回x的正切值 |

数组篇

Array 类型

常用

每个位无类型限制

大小是动态调整的

创建方式:

```
1. var color = new Array();
```

```
Var colors = new Array(20); (注意: var color = new Array("20")—
```

只有一项的数组)

```
Var colors = new Array{"red", """};
```

实际上, new 是可以省略的

```
Var colors = Array(5);
```

2. 数组字面量表示法

```
Var arr =[1,2,3,5];
```

数组的 length 属性

Arrr.Length—数组长度

特殊用途:

1. 删除元素。

```
Arr.length = Arr.length - 1;
```

2. 添加元素

```
Arr [ arr.length +1 ] = value;
```

实际上, 添加时下标是任意的, 可以很大, 数组大小是最大的下标+1

遍历数组

```
for(var i =0;i<a.length;i++){  
    document.write(a[i]+"<br>");  
}
```

注意:

For-each 语句不起作用

```
for( i in a){  
    document.write(i+"<hr>");  
    //输出的实际上是定义的非 undefined 的下标  
}
```

数组检测之 isArray()

```
if (Array.isArray(arr)) {  
    alert("arr is a array!");  
}
```

转换方法

一切对象都有的方法:

- 1.toLocaleString()
- 2.toString();
- 3.valueOf()

调用数组的 toString 返回一个有逗号拼接的字符串

调用 valueOf 返回的还是数组

toLocaleString()也返回字符串

数组的栈方法

1.push()

返回压入栈后元素的总数目

```
Arr.push(1,2,3);
```

2.pop()

出栈，返回最后一个元素

数组的队列方法

1.shift();

获取最前面的一个元素并返回

```
Arr.shift();
```

2.push();

从尾端进队列

数组重排序方法

两种方法都会改变数组的值，若需多次利用，备份之

1.reverse()

反转数组顺序

2.sort();

默认按升序排序

比较原理：

调用元素自身的 `toString` 方法将元素转化为字符串，然后按字典序排列

就是是数字，也会先转化为字符串。所以无法进行数字的降生排序。

需要则添加比较函数

比较函数

参数是两个值

返回类型有三种：

第一个在前，返回值大于 0

二者同样，返回 0；

第一个在后，返回值小于 0

Eg：升序的比较函数：

```
function comp(a,b){  
    return a-b;  
}
```

将函数名作为参数传入 `sort()`；

```
a.sort(comp);
```

数组操作方法

1.concat()

参数是一个数组：创建一个副本

参数是多个数组：创建一个副本，将几个数组连接起来。

参数不是数组：创建一个空副本，将元素按顺序添加末尾

(实际上：先创建一个空副本，按顺序添加即可)

```
var a_copy = a.concat();
```

```
var ab_copy = a.concat(b);
```

```
var abc_copy = a.concat(b,c);
```

```
var d = a.concat('new','new1','new2','new3',b);
```

2.slice ()

数组拆分合成并返回，接受 2 个参数，完全同于 Java 的 subString();

1.slice(index)

返回 index 即以后所有项

2.slice(first, last)

返回[first, last)之间的元素

注意:

1.有负数下标，则加上数组长度生成新的下标，即取倒数

2.结束位置小于起始位置，返回空数组

3.splice()—最强大

主要用途：向数组中部插入项

1.删除(2 参数): splice(delete_begin_index, delete_numbers)

2.插入(3+参数): splice(delete_begin_index, delete_numbers, insert_contents)

参数：插入/删除起始位置、要删除的数目、插入内容（至少一个）

Eg: a.splice(0,0,'red','gg');

3.替换(用插入思想实现)

先删除，再插入

数组定位方法

1.indexOf(target, begin_index) :从前往后

2.lastIndexOf(target, begin_index) : 从后往前

查找不到, 返回-1

查找条件: 严格相等, 即===

找到后返回第一个的下标

数组迭代方法

五个迭代方法格式: func(f(item, index , arraySelf), 运行函数的作用域对象(可选)).

函数是有返回值的, 返回布尔值或其他, 其实也可无

(实际上, f 的参数也不是固定的)

1.every(f(),)—f()返回布尔值

F()对于每个数组元素都返回 true 时才返回 true, 否则返回 false

2.filter (f(),) —f()返回布尔值

返回运行函数后返回 true 的元素组成的子数组

3.forEach(f(),)—f()返回运算后的 item

对每个元素运行函数, 无返回值

4.map(f(),)—f()返回运算后的 item

每个元素运行函数, 返回运行后产生结果组成的数组

5.some(f(),)—返回布尔值

与 every 相似, 一个满足即返回 true

注意: 要想返回值都被操作过, 定义的函数必须有返回值:


```
var a =[1,2,3,4,5];  
var res = a.map(function (value, index, array)  
    return array[index]*10;  
});  
document.write(res);|
```

数组的缩小方法

1.reduce(f(), 可选的作为缩小基础的初始值)

从头到尾，遍历所有项

2.reduceRight(f() , 可选的作为缩小基础的初始值)

从未到头，遍历所有项

F():

f(前一个值 prev , 当前值 cur, 索引 index , 数组对象 array){}

eg:

```
var v = [1,2,3,4,5,6];  
var sum = v.reduce(function(prev,cur, index,array){  
    return prev+cur;  
})  
show(sum); //21;
```

join()

arr.join()方法可以把一个数组的所有元素转换成字符串，然后再把他们连接起来，可以添加分隔符，如空串“”，默认是逗号

```
var b = ["A", " k" ,"i ", "r ", "a "];  
var name = b.join(""); //name 的值是 Akira
```

时间

Date 类型

使用 UTC—1970.1.1 零时开始以毫秒计时

建立时间对象：

1. 不传参数：`var now = new Date();`

自动获取当前时间。

2. 传递参数：可获取指定时间

`Var time = new Date(utc 毫秒数 m);`

获取毫秒数的方法：

a. `Date.parse(多种格式时间)` 返回毫秒数

时间格式：月/日/年

英文月 日,年

英文星期 英文月 日 年 时:分:秒 时区

也可以直接将这些格式的时间传入

`Var time = new Date(1/2/2019);`

b. `Date.UTC()` 返回毫秒数

参数：

年份：必须

月份：基于 0,0---11，必须

月中哪天：1-31，可省，省则默认 1

小时数：0-23，省则为 0

分、秒、毫秒：省则为 0

时间对象可以调用一系列方法：如 getDate() getYear()

时间格式化方法

toDateStr() 显示星期、月、日、年

toTimeString() 显示时、月、日、年

toLocaleDateString() 显示：特定地区格式下星期、月、日、年

toLocaleTimeString()

toUTCString

时间/日期组件方法

正则篇

语法: `var exp = /pattern/flags;`

Pattern : 模式串

Flags : 标志, 可以组合

1. **g** : 全局模式 **global**, 被应用于所有字符串
2. **i** : **case-insensitive** 不区分大小写
3. **m**: **multiline** 多行模式, 到达一行末时还会查找下一行

元字符: { `[\ ^ $ | () ? * + .]` } 需要转义

Eg: 匹配所有‘at’, 不区分大小写

```
Var pat = /at/gi;
```

构造正则表达式

1. `var number = new RegExp(something);`
- 2.

```
var Pattern = {};  
Pattern.empty = /^[s\n\r\t]*$/;  
Pattern.RegInt = /^[0-9]*[1-9][0-9]*$/; //整数  
Pattern.RegFloat = /^[+]?0|([1-9][0-9]*) ([.] [0-9]+)?$/; //浮点数  
Pattern.RegMoney = /^[+]?0|([1-9][0-9]*) ([.] [0-9]{1,2})?$/; //货币  
Pattern.RegSPhone = /^[0-9]{6,8}([-][0-9]{1,6})?$/; //电话号码(短)  
Pattern.RegLPhone = /^[0-9]{3,4}([-][0-9]{6,8})([-][0-9]{1,6})?$/; //电话号码(长)  
Pattern.RegCellPhone = /^[0-9]{11}$/; //手机号码  
Pattern.RegEmail = /^[w+([-+.]w+)*@w+([-+.]w+)*\.w+([-+.]w+)*$/; //电子邮件  
Pattern.RegURL = /^http:\/\/([w-]+\.)+[w-](\/[w- .\/?%&=]*)?$/; //网页地址
```

| 表 10.1 字符的转义规则 | |
|----------------|----------------------------|
| 字 符 | 匹 配 |
| 字母数字字符 | 自身 |
| \0 | NUL 字符(\u0000) |
| \t | 制表符(\u0009) |
| \n | 换行符(\u000A) |
| \v | 垂直制表符(\u000B) |
| \f | 换页符(\u000C) |
| \r | 回车(\u000D) |
| \xnn | 由十六进制数 nn 指定的 Latin-1 字符 |
| \uXXXX | 由十六进制数 xxxx 指定的 Unicode 字符 |
| \cX | 控制字符^X。 |

范围定义： `[0-9a-zA-Z]`

布尔非： `[^abc]`——非 abc

| 字 符 | 匹 配 |
|---------------------|---|
| <code>[...]</code> | 位于括号之内的任意字符 |
| <code>[^...]</code> | 不在括号之中的任意字符 |
| <code>.</code> | 除换行和 Unicode 行终止符之外的任意字符 |
| <code>\w</code> | 任何 ASCII 单字字符，等价于 <code>[a-zA-Z0-9_]</code> |
| <code>\W</code> | 任何非 ASCII 单字字符，等价于 <code>[^a-zA-Z0-9_]</code> |
| <code>\s</code> | 任何 Unicode 空白符 |
| <code>\S</code> | 任何非 Unicode 空白符 |
| <code>\d</code> | 任何 ASCII 数字，等价于 <code>[0-9]</code> |
| <code>\D</code> | 除了 ASCII 数字之外的任何字符，等价于 <code>[^0-9]</code> |
| <code>\b</code> | 退格直接量 |

重复：

`*`： `0+`

`+`： `1+`

`?`： `0` 次或次重复

`{n}`： 匹配一个字符 `n` 次

`{n,m}`： 匹配 `n-m` 次

分组： 用括号分组，

引用： 使用“`?`”生成引用

指定匹配位置:

`^` 匹配字符串的开头, `/^Script/` 不与 `JavaScript` 匹配 (注意: 此处 `^` 并非布尔非)

`$` 匹配字符串的结尾, `/\.js$/` 不与 `dfsd.jsdd` 匹配 (`$` 用于末尾处, `^` 在开头)

`\b` 匹配单词的边界, `\B` 匹配非单词边界

`/^[Jj]ava[Ss]cript\b/` 与 “JavaScript is fun!” 匹配, 不与 “JavaScript:alert(0);” 匹配。

如果你用 `(?!和)` 符号, 那么它是一个反前向声明, 含义与前向声明相反

模式匹配

用于 `String`:

`Search()`
`Replace()`
`Match()`
`Split()`

用于模式匹配的 `RegExp` 方法

`exec(str)`:

类似 `String` 的 `match`, 匹配失败, 返回 `null`, 匹配成功, 返回一个数组, 第

一个元素是匹配项, 剩下的元素是其余匹配的各个分组

`Test(str): pattern.test(string)`

如果匹配, 返回 `true`

`RegExp` 属性

正则表达式对象(不管是直接量还是由 RegExp 构造的对象)拥有五个实例属性,它们分别是 source、global、ignoreCase、lastIndex 和 multiline。

source 属性是一个只读的字符串属性,它包含了描述这个正则表达式的文本。

global 属性是一个只读的布尔型属性,它表明了这个正则表达式是否具有“g”标志。

ignoreCase 属性是一个只读的布尔型属性,它表明了这个正则表达式是否具有“i”标志。

multiline 属性是一个只读的布尔型属性,它表明了这个正则表达式是否具有“m”标志。

lastIndex 属性是一个可读写的属性,它的作用在前面已经介绍过了,当正则表达式拥有“g”标志的值时,这个属性表明了下一次检索字符串时的起始位置。

面向对象程序设计

对象创建注意

```
getName:function () {  
    return person.name;  
},  
getName1: function(){  
    return this.name; //必须使用this  
},  
getName2:function(){  
    return name;// 错误  
},
```

属性类型

ECMA 有 2 种属性: 数据属性 访问器属性

数据属性

1. [configurable] : 表示能否通过 delete 删除属性从而重新定义属性,能否修改属性特性,能否把属性修改为访问器属性。

默认值: true

2.[`Enumerable`] : 表示能否通过 `for-in` 循环返回属性。

默认: `true`;

3.[`Writable`] : 可修改性

默认值: `true`

4.[`value`] : 包含该属性的数据值。

默认值: `undefined`

修改特性的方法:

`Object.defineProperty(属性所在对象, 属性名, {描述符对象});`

其中描述符 `descriptor` 对象属性: `configurable enumerable writable value`

```
var god = {  
  name: 'Xutao'  
};  
// var god = {};  
Object.defineProperty(god, 'name', {  
  writable: false,  
  value: "xutao'gf",  
  configurable: false,  
  enumerable: false  
});  
show(god.name);  
god.name = 'changed';  
show(god.name+" | didn't change");
```

访问器属性

包含函数 `getter` `setter`, 非必须

读取访问器: 调用 `getter`

写入访问器：调用 setter

1.configurable

2.Enumerable

3.Get 默认 undefined

4.Set 默认 undefined

同样使用 Object.defineProperty()定义

```
var book = {
  name: 'love-teaching'
};
Object.defineProperty(book, name, {
  get: function () {
    return this.name;
  },
  set: function f(newValue) {
    this.name = newValue;
  }
});
// book.set('this is a new book!'); 错误，不能手动调用。
book.name = "this is new!"
show(book.name);
```

Object.definePorperties—定义多属性

Object.definePorperties(目标对象, {属性: {name: value}})

```

var book = {
  name: 'love-teaching',
  Edition: '2019-2-2'
};
Object.defineProperty(book, {
  name: {
    value: 'changed',
    // set: function (v) { 错误，访问器属性和数据属性不能混合定义
    //     this.name = v;
    // }
  },
  Edition: {
    value: '1997-2-2',
    // get: function () {
    //     return this.Edition;
    // }
  },
  visitPropererty: { //访问器属性，
    set: function (v) {
      this.name = v;
    },
    get: function () {
      return this.name;
    }
  }
});

```

读取属性的特性

`Object.getOwnPropertyDescriptor(对象, "对象属性");`

```

var descriptor = Object.getOwnPropertyDescriptor(book, "_year");
alert(descriptor.value); //2004
alert(descriptor.configurable); //false

```

创建对象

工厂模式

工厂模式：一种设计模式，抽象了创建具体对象的过程

ECMA 中无法创建类，用一种函数来封装以特定接口创建对象的细节。

```
function createPerson(name,age) {
    var o = new Object();
    o.name = name;
    o.age = age;
}
var o = createPerson('xutao',121);
```

构造函数模式

```
function Person(name,age,isSingle) {
    this.name = name;
    this.age = age;
    this.isSingle = isSingle;
    this.getInfo = function () {
        return [this.name,this.age,this.isSingle];
    }
}
var p = new Person('Edwin xu',21,true);
show(p.getInfo());
```

注意:

函数名一般使用大写字母开头，使用 new 实例化对象

p.constructor 即可返回构造函数

函数定义也可以用: this.sayName = new Function(“function content”)

也可以将使用外部函数 this.func = externalFunc;

原型模式

每个函数都有一个 prototype（原型）属性（是一个指针，指向一个对象）

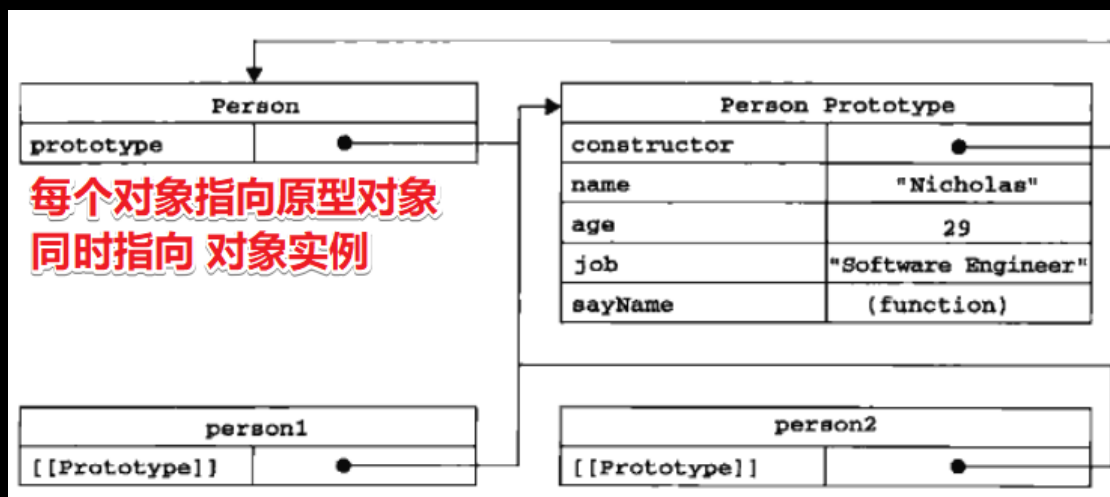
Prototype 通过调用构造函数而创建的那个对象实例的原型对象

原型对象优点：让所有对象实例共享他所包含的属性和方法

每个原型实例对象都有 constructor 属性，指向构造函数

```
//原型对象
function Dog(name,age) {
  // this.name = name;
  // this.age = age;
}
Dog.prototype.name = 'Edwin';
Dog.prototype.age = 21;
Dog.prototype.sayName = function () {
  show(this.name);
};
```

Obj.prototype.key = value 构造原型



原型对象的 isPrototypeOf() 方法

```
show(Dog.prototype.isPrototypeOf(p)); //false
show(Dog.prototype.isPrototypeOf(d)); //true
```

格式:

Obj.prototype.isPrototypeOf(实例化对象)

注意:

原型对象中的值是不能改变的，只能改变实例化的对象的值，实例化对象的值也是来源于原型，可以用新值屏蔽

方法: Obj.hasOwnProperty('属性')—检测一个属性是否存在对象实例中。

原型与 in 操作符

(in 有 2 种使用：1.foreach ； 2.单独使用)

格式：‘属性’ in 对象实例—返回 Boolean

```
show('name' in hellokitty); //true
```

Object.keys()方法

用途：获取对象上所有可枚举的实例属性

```
show(Object.keys(Dog.prototype)); //name,age...  
show(typeof Object.keys(Dog.prototype)); //object  
show(Object.keys(Dog.prototype)[1]) //age //类array
```

原型语句简化

```
function Fish() {  
}  
Fish.prototype = {  
  name: 'GoldFish',  
  age: 1,  
  taste: 'delicious'  
};
```

实例中的指针仅指向原型，而不指向构造函数。

```
function T() {  
}  
// var t = new T(); t.sayIt() error  
T.prototype = {  
  name: 'XT',  
  age: 21,  
  sayIt: function () {  
    show("say it!");  
  }  
};  
var t = new T(); // t.sayIt() no problem  
t.sayIt(); //
```

原生对象原型

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法。

在一个已存在的对象构造器中是不能添加新的属性的

所有的 JavaScript 对象都会从一个 prototype（原型对象）中继承属性和方法

所有 JavaScript 中的对象都是位于原型链顶端的 Object 的实例。

JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

使用 prototype 属性就可以给对象的构造函数添加新的属性：

继承

继承种类：接口实现、实际继承

ECMA 只支持实际继承，继承方式：原型链

函数表达式

创建函数

1. 常规: `function a(){}`

2. 函数表达式，即匿名函数 anonymous Function

```
var func = function (a,b) {  
    return a*b;  
};  
show(func(12,21)); //252
```

函数提升：

在函数执行前，会先识别函数定义，故函数定义可在执行语句后

但是：函数表达式方式定义的函数不具有此特性

闭包

官方定义：一个拥有许多变量和绑定了这些变量的环境的表达式（通常是函数）

特点：

1. 作为一个函数变量的一个引用，当函数返回时其处于激活状态。

2. 一个闭包就是当一个函数返回时，一个没有释放资源的栈区

闭包：有权访问另一个函数作用域的变量的函数，即函数中的函数在该函数外被调用时就会形成闭包

理解闭包，首先必须理解 Javascript 特殊的变量作用域。

变量的作用域无非就是两种：全局变量和局部变量。

Javascript 语言的特殊之处，就在于函数内部可以直接读取全局变量。

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。

```
function f1(){
    var n=999;
    nAdd=function(){n+=1}
    function f2(){
        alert(n);
    }
    return f2;
}
var result=f1();
result(); // 999
nAdd();
result(); // 1000
```

在这段代码中，result 实际上就是闭包 f2 函数。它一共运行了两次，第一次的值是 999，第二次的值是 1000。这证明了，函数 f1 中的局部变量 n 一直保存在内存中，并没有在 f1 调用后被自动清除。

为什么会这样呢？原因就在于 f1 是 f2 的父函数，而 f2 被赋给了一个全局变量，这导致 f2 始终在内存中，而 f2 的存在依赖于 f1，因此 f1 也始终在内存中，不会在调用结束后，被垃圾回收机制 (garbage collection) 回收。

使用闭包的注意点

1) 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在 IE 中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2) 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象 (object) 使用，把闭包当作它的公用方法 (Public Method)，把内部变量当作它的私有属性 (private value)，这时一定要小心，不要随便改变父函数内部变量的值。

创建闭包

常见方式：在函数内部创建函数

```
//第4种写法
var Circle={
    "PI":3.14159,
    "area":function(r) {
        return this.PI * r * r;
    }
};
alert( Circle.area(1.0) );
```

闭包与变量

~~闭包只能取得包含函数中任何变量的最后一个值，~~ 现在似乎不行

BOM

BOM: 浏览器对象模型

Window 对象

BOM 核心对象是 window，它是浏览器的一个实例，HTML 中所有的属性和变量、函数，都是 window 的 Global 对象

```
for (i in window){  
    show(i);  
}
```

输出当前窗口的属性

Window 对象声明周期

一个 window 对象在某个浏览器进程被终止前总是存在的，即使你关闭这个窗口

注意: window 对象总是引用当前窗口的 window 对象，

窗口关系及框架

页面包含框架，每个框架都拥有自己的 window 对象，并且保存在 frames 集合

中，可以通过下标索引或框架名访问。每个框架都有一个 name 属性

Window 属性及方法

| 名 称 | 类 型 | 说 明 |
|-------------------|-----|---------------------|
| Alert | 方法 | 弹出简单提示框 |
| AttachEvent | 方法 | 注册事件 (IE 有效) |
| Blur | 方法 | 窗口失去焦点 |
| ClearInterval | 方法 | 停止计数器运行 |
| ClearTimeout | 方法 | 取消定时器 |
| clientInformation | 属性 | 客户端信息 |
| clipboardData | 属性 | 剪贴板数据 |
| close | 方法 | 关闭当前窗口 |
| closed | 属性 | 当前窗口是否关闭 |
| confirm | 方法 | 确认对话框 |
| createPopup | 方法 | 构造弹出窗口 (IE 有效) |
| defaultStatus | 属性 | 默认状态栏 |
| detachEvent | 方法 | 取消注册的事件 (IE 有效) |
| dialogArgument | 属性 | 模态对话框参数 (IE 有效) |
| dialogHeight | 属性 | 模态对话框高度 (IE 有效) |
| dialogLeft | 属性 | 模态对话框左上角横坐标 (IE 有效) |
| dialogTop | 属性 | 模态对话框左上角纵坐标 (IE 有效) |
| dialogWidth | 属性 | 模态对话框宽度 (IE 有效) |
| document | 属性 | 当前窗口的 Document 对象引用 |
| event | 属性 | 事件参数 (IE 有效) |
| execScript | 方法 | 执行脚本 (IE 有效) |
| external | 属性 | 浏览器扩展对象 (IE 有效) |
| focus | 方法 | 窗口获得焦点 |
| frameElement | 属性 | 窗口框架元素的引用 |
| frames | 属性 | 当前窗口中的框架 |
| history | 属性 | 当前窗口中的 History 对象引用 |
| images | 属性 | 当前窗口中的图片引用 |

| 名 称 | 类 型 | 说 明 |
|--------------------|-----|----------------------|
| length | 属性 | 当前窗口中框架的数量 |
| location | 属性 | 当前窗口中的 Location 对象 |
| menuArguments | 属性 | 上下文菜单参数 |
| moveBy | 方法 | 移动窗口 |
| moveTo | 方法 | 移动窗口 |
| name | 属性 | 当前窗口的名字 |
| navigate | 方法 | 浏览某个 URL |
| navigator | 属性 | 当前窗口的 Navigator 对象引用 |
| open | 方法 | 打开新窗口 |
| opener | 属性 | 父窗口引用 |
| option | 属性 | 下拉列表选项构造器 |
| parent | 属性 | 当前窗口的父框架 |
| print | 方法 | 打印当前窗口中的文档内容 |
| prompt | 方法 | 询问对话框 |
| resizable | 属性 | 窗口是否允许改变大小 |
| resizeBy | 方法 | 改变窗口大小 |
| resizeTo | 方法 | 改变窗口大小 |
| returnValue | 属性 | 设置窗口返回值 (IE 有效) |
| screen | 属性 | 当前窗口的 Screen 对象引用 |
| screenLeft | 属性 | 屏幕偏移量 |
| screenTop | 属性 | 屏幕偏移量 |
| scroll | 方法 | 控制滚动条 |
| scrollBy | 方法 | 控制滚动条 |
| scrollTo | 方法 | 控制滚动条 |
| self | 属性 | 当前窗口自身的引用 |
| setInterval | 方法 | 设置计数器 |
| setTimeout | 方法 | 设置定时器 |
| showHelp | 方法 | 显示一个帮助文档 (IE 有效) |
| showModalDialog | 方法 | 模态对话框 (IE 有效) |
| showModelessDialog | 方法 | 非模态对话框 (IE 有效) |
| status | 属性 | 状态栏 |
| top | 属性 | 当前窗口的最上层框架的引用 |
| toString | 方法 | 转为字符串 |
| unadorned | 属性 | 是否对窗口进行修饰 (IE 有效) |
| window | 属性 | 当前窗口自身的引用 |

一个例子:

```

INPUT IT:<input type="text" value="input..." id="text">
<button onclick="openNewWin()">CLICK ME</button>
<script type="text/javascript">
    function openNewWin() {
        var w = open();
        var d = document.getElementById('text');//获取页面输入框
        var cont = d.value;//获取输入框内容
        with(w){
            document.writeln("<textarea style='width: 400px,height: 200px;background-color:yellowgreen;'>" + cont + "</textarea>");
            document.writeln("<br><button onclick='self.close()'>Good Bye!</button>"); //调用self.close()关闭当前窗口。
        }
    }
</script>

```

对话框与状态栏

对话框

- `Alert ()`

只接受一个参数

- 对话框 2-`confirm()`

Yes---返回 `true`

No---返回 `false`

- `Prompt()`

输入框，可以嵌套在其他语句中。

如果点击了取消，返回 `null`

注意：这三个都是进程同步对话框，一旦执行，程序会停止等待，直到交互结束

模态/非模态对话框

模态对话框 `modal`：焦点锁定的对话框，不能点击其他对话框

非模态对话框 `modeless`：焦点不会锁定，如 `window.open()`

状态栏

- `Window.status`
- `Window.defaultStatus`

框架——上层 Window 对象

`Frameset`、`frame` 似乎被 HTML5 舍弃了

Iframe: 内联框架

Frames 属性

每一个 window 对象都有一个 frames 属性

注意: Frame 是可以递归嵌套的, 如: `window.frames[0].frames[1]`

Window.parent

每一个 window 对象有一个 parent 属性, 引用父 window, 若无父 window, 则为自身

Window.self

Self 引用自身, 指向当前框架,

window.top

指向顶层的框架

框架的命名

Frame/iframe 都含有一个 name 属性, 指定框架的名字

```
<body>
  <!--a标签跳转-->
  <!--a链接标签target指向一个框架时, 可以将链接内容显示在该框架中-->
  <a href="http://www.baidu.com" target="myF">Frame</a>
  <iframe src="" frameborder="1px" name="myF"></iframe>

  <!--方法2: frame.location = targetUrl-->
  <a href="#" onclick='a.frame.location="http://www.baidu.com"'>click me</a>
  <iframe name = 'a frame' src="" frameborder="3px"></iframe>
</body>
```

页签

类 GUI, 复杂界面分页展示的交互界面

在 WEB 中, 用层和框架模拟页签, 用 js 控制页签增加删除

未完待续。。。

表单-表单对象

Document.forms

Document.forms 存放的是表单集合，他们是按页面中出现的先后顺序存放的。

```
document.forms[document.forms.length - 1]
```

Form 对象属性

- Elements[]集合，包含 JS 的各种表单对象，按先后顺序

可以通过每个对象的 value,name 等属性访问

- Action
- Encoding
- Method
- Target

| 类 型 | 说 明 | 类 型 | 说 明 |
|----------|------|----------|---------|
| button | 按钮 | radio | 单选框 |
| checkbox | 复选框 | reset | 重置按钮 |
| file | 文件 | submit | 提交按钮 |
| hidden | 隐藏元素 | text | 文本输入框 |
| image | 图片 | textarea | 多行文本输入框 |
| password | 密码框 | | |

正则+表单校验

可以将模式作为表单属性

DOM

Document 对象

Document 对象—代表浏览器窗口的文档内容。

DOM: 文档对象模型, Document 对象及它呈现给 js 程序的元素集合构成了 DOM

概览

浏览器窗口装载一个新的页面时, 总是初始化一个新的 Document 对象。

Document 函数

Document.title = "new title" 修改标题

```
document.title = "this is a new title!";
document.write("document.write");
document.writeln("document.writeln");
// document.open();//打开文档
// document.close();//关闭文档
```

注意:
才是 HTML 的换行符, writeln () 输入的换行“\n”并不能显示在页面中。

Document . images—枚举出页面文档中所有的图片, 返回集合。

Document 对象基本信息

- title: html 中<title>标签中的名称, Document.title = “newTitle”
即可更改标题

- `domain` 属性：使处于同一 Internet 域中的相互信任的 WEB 服务器交互时能协同地放松某项安全限制。
- `referrer` 属性：把浏览器带到当前文档的链接，如果当前页面不是从其他页面跳转过来的，`referrer` 属性为空字符串。注意：这是只读的，不可写。
- `lastModified` 属性：文档最近修改日期字符串。

Document 对象外观属性

- `linkColor` 未被访问的链接颜色 舍弃
- `vlinkColor` 被访问过的链接的正常颜色 舍弃
- `alinkColor` 被激活的链接颜色 舍弃
- `fgColor` 舍弃
- `bgColor` 舍弃
- 全舍弃

子对象接口

```
document.anchors; //Anchor对象一个结合，代表文档中锚
document.applets; //Applet对象的一个集合，代表Java小程序
document.forms; //Form对象的一个集合，代表文档中的表单元素。未舍弃
document.images; //文档中的图片集合
document.links; //Link对象的集合，代表文档的链接
```

```
var a = document.links; //Link对象的集合，代表文档的链接
for (i in a){
    document.writeln("</br>" + i); //显示links属性
}
for (var i = 0; i < a.length; i++){
    document.writeln("</br>" + a[i]); //显示链接内容： |
}
```

例子-反转图片：

```
function reversePictures() {
    var image = document.images;
    for (var i =0; i<image.length;i++){
        image[i].style.filter="FlipV";
    }
}
```

Document.cookie—非集合，为脚本提供有限的数据存储能力

正则与表单校验

Pattern 可以是表单的属性：

电话号码输入：<input type="text" name="b" pattern=/\d{11}/>

通过调用来校验

内置对象-Navigator

window.navigator 引用一个 Navigator 对象，Navigator 对象提供的是浏览器信息

Navigator 对象属性

vendorSub
productSub
vendor
maxTouchPoints
hardwareConcurrency
cookieEnabled
appName—web 浏览器名称
appVersion

platform
product
userAgent
language
languages
onLine
doNotTrack
geolocation
mediaDevices
connection
plugins
mimeTypes
webkitTemporaryStorage
webkitPersistentStorage
serviceWorker
getBattery
sendBeacon
getGamepads
getUserMedia
webkitGetUserMedia
javaEnabled
vibrate
requestMIDIAccess
budget
permissions
presentation
registerProtocolHandler
unregisterProtocolHandler
deviceMemory
clipboard
credentials
storage
keyboard
usb
requestMediaKeySystemAccess
mediaCapabilities

内置对象-Screen

Screen 对象—提供显示器分辨率和可用颜色数信息

属性：

availWidth
availHeight
width
height
colorDepth
pixelDepth
availLeft
availTop
orientation

内置对象-Location

Location—当前窗口中显示文档的 URL 代表

属性：

replace
assign
href—完整的 URL
ancestorOrigins
origin
protocol
host
hostname
port
pathname
search—? 号后的搜索内容
hash
reload
toString

重定向方法：

1.reload()—当前页面文档重新加载

2.replace(url)—使用 URL 指向的页面来替换当前页面

```
<button onclick="window.location.reload()">重加载</button>  
<button onclick="window.location.replace('form.html')">替换</button>
```

内置对象—History

浏览器记录，很少使用

Window 重要属性

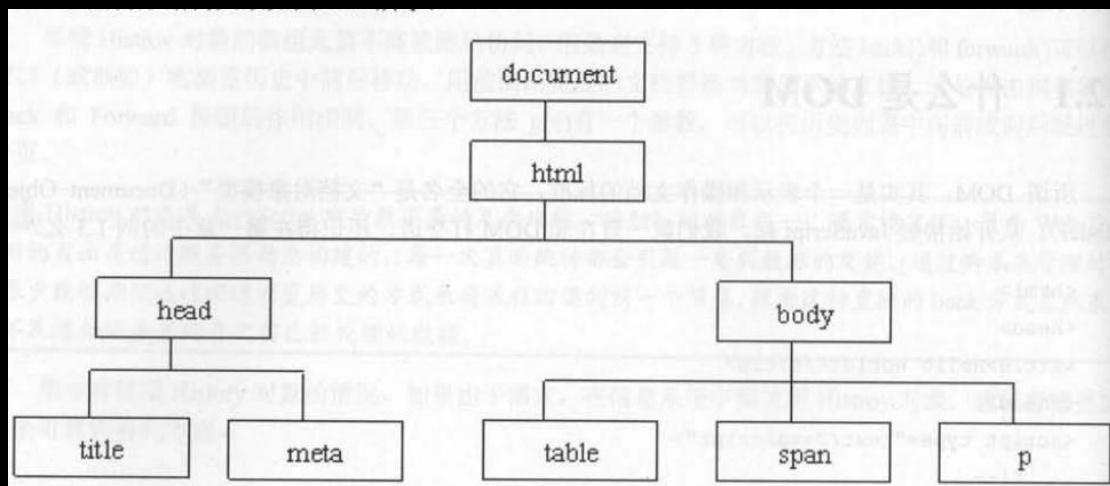
- onclick—点击
- onfocus—获取焦点
- onmouseover—鼠标箭头在上
- onmouseout—鼠标箭头不在其上
-

DOM高级

文档—树模型

Html 文档的递归形式决定了其树型拓扑结构

节点



文档中不同的结构类型分别对应不同类型的 DOM 节点，每个节点有属性

`nodeType`，指定节点的类型。

常见节点类型：

| 整数值 | 节点类型常量 |
|-----|-----------------------------|
| 1 | ELEMENT_NODE |
| 2 | ATTRIBUTE_NODE |
| 3 | TEXT_NODE |
| 4 | CDATA_SECTION_NODE |
| 5 | ENTITY_REFERENCE_NODE |
| 6 | ENTITY_NODE |
| 7 | PROCESSING_INSTRUCTION_NODE |
| 8 | COMMENT_NODE |
| 9 | DOCUMENT_NODE |
| 10 | DOCUMENT_TYPE_NODE |
| 11 | DOCUMENT_FRAGMENT_NODE |
| 12 | NOTATION_NODE |

DOM 对象的通用属性/方法

- `ChildNodes`：子节点集合，类型：`NodeList`，非数组
- `FirstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`
- `parentChild`

- tagFather.appendChild(TagChild)—附加到某标签的子标签末尾
- TagFather.removeChild(TagChild)
- TagFa.Replace(TagChild)
- TagFa.insertBefore(Child)

DOM 之 HTML 方法

- createElement(newEle_str)

```
//先创建一个元素
var div1 = document.createElement('div');
//设置它的内部HTML内容
div1.innerHTML = "<h1 align='center'>从JS创建的H1</h1>";
/**必须将创建的标签添加到目标标签的子Tag
document.getElementsByTagName('body')[0].appendChild(div1);
```

- setAttribute(标签属性名: 值)
- getAttribute(属性名)
- removeAttribute()

```
var div1 = document.createElement('div');
//设置属性
div1.setAttribute('name','div_name');
//获取属性
show(div1.getAttribute('name'));
```

- outerHTML 属性可以查看一个 DOM 元素完整的 HTML 文本

```
if (document.body.outerHTML){
    alert(document.body.outerHTML);
}
```

DOM 与浏览器实现

不同浏览器对 DOM 有不同的实现

设置 DOM 对象属性, 除了通过 setAttribute(), 还能直接设置, 如:


```

Div1.title =
Div1.id =

Div1.style.color = 'red' //通过 style 属性来设置具体属性

```

➤ 页面元素树输入:

```

//转化为浏览器输入: |
var s = "";
function travel(space,node) {
    if (node.tagName) {
        s+=(space+node.tagName+"<br>");
    }
    var L = node.childNodes.length;
    for (var i =0;i<L;i++){
        travel(space+"|-",node.childNodes[i]);
    }
}
travel("",document);
document.write(s);

```

表 12.2 DOM 对象和 HTML 节点类型的对应关系

| HTML 节点类型 | HTML 成员 | DOM 对象 | 实现接口 |
|-----------------------------|--|------------------|----------------------------------|
| Node.ELEMENT_NODE | <head><body><a> <div> <table><form>... | Element | Node Element HTMLElement |
| Node.TEXT_NODE | 标记之间的文字内容 | Text | Node CharacterData Text |
| Node.DOCUMENT_NODE | 无 | Document | Node Document HTMLDocument |
| Node.COMMENT_NODE | <!--与-->之间的注释 | Comment | Node CharacterData Comment |
| Node.DOCUMENT_FRAGMENT_NODE | HTML 文档片断 | DocumentFragment | NodeSet |
| Node.ATTRIBUTE_NODE | 标记的属性 | Attribute | Node Attr |

| 表 12.3 DOM 接口 | |
|---------------|----------------------|
| HTML 成员 | DOM 接口 |
| <head> | HTMLHeadElement |
| <body> | HTMLBodyElement |
| <title> | HTMLTitleElement |
| <p> | HTMLParagraphElement |
| <input> | HTMLInputElement |
| <table> | HTMLTableElement |

创建/删除节点

创建新节点

- `Document.createElement()`—创建 Element 节点
- `Document.createTextNode()`—创建 text 节点
- `Document.createAttribute()`—创建 attribute 属性节点

通过文档元素直接创建

设置 `innerHTML` 属性来创建元素

优点：创建多层次的 HTML 文档时，要比 `append()\insertBrfore()` 快

```
var d = document.getElementById('div1');
d.innerHTML = "<button>CLICKME</button><br/>输入: <input value='input...'>";
```

(`outerHTML` 表示包括自身在内的所有子集 HTML 内容)

注意：

```
document.write(d.outerHTML);
```

Document 写入时会直接显示出效果，而不是 text 文本内容，使用 `alert` 才会

显示 text 文本形式

故可以使用这种方式重用代码，复用形式，及其高效

访问\操纵 DOM 节点

遍历节点

```
var elementName = ""; //用来存储所有的节点名
function countElements(node) {
    var count = 0;
    if (node.nodeType==1){
        count++;
        elementName = elementName+node.tagName+"<br/>";
    }
    for (var m = node.firstChild;m!=null;m=m.nextSibling){ //关键代码，递归进行。
        count+=countElements(m);
    }
    return count;
}
document.write("元素数合计="+countElements(document)+"<br/>"); //使用Document切入，因为Document是所有文本元素的父亲
document.write(elementName);
```

```
var s = '';
function print(space,node) {
    if (node.tagName){
        s+=(space+node.tagName+"<br>");
    }
    var L =node.childNodes.length;
    for (var i=0;i<L;i++){
        print(space+"|-",node.childNodes[i]);
    }
}
print("",document);
document.write(s);
```

打印递归树

获取节点的方法

- getElementById(): 唯一性，不存在则返回 null
- getElementsByTagName(): 按标记名字，返回一个集合，无则集合长为 0
- getElementsByName(): 按名字，返回相同名字到集合，无则长为 0

注意：上述三个搜索方法承接于 Document 接口，接口 Element 也含有一个 getElementByTagName(), 不同的是，该方法只在当前节点及其子节点中搜索，而不是在全局 Document 中搜索

克隆节点

克隆方法: node.cloneNode(boolean)

`node.cloneNode(true)`—表示克隆 `node` 及其所有子节点

`false` 则是只克隆当前节点

注意：克隆节点后，需要添加到预定父节点后

如：列表添加：

```
function addOne() {  
    var a = document.getElementById('t5');  
    var a_t = a.cloneNode(true);  
    var t = document.getElementById('t');  
    t.appendChild(a_t);  
}
```

添加/移除/替换节点

- `appendChild(c)`—添加节点
- `removeChild(node)`—移除节点
- `replaceChild (newNode, oldNode)` —替换节点
- `insertBefore(newNode, oldNode)`—将新节点插入旧节点之前, `oldNode` 若是缺省，则同 `appendChild`。

外观与行为

`Style` 属性控制许多属性

显示与隐藏

隐藏元素的方法

1. `opacity = 0` 透明度设为 0，视觉上隐藏
2. `visibility = hidden` 还会占据原来的位置
3. `display=none` 不占空间，（有碍于交互）

4. position 移到不可视区域

色变

```
//颜色切换
function changeColor() {
    var temp = document.getElementsByTagName('body')[0];
    var R = Number(Math.random()*256);
    var G = Number(Math.random()*256);
    var B = Number(Math.random()*256);
    temp.style.backgroundColor = "rgb("+R+", "+G+", "+B+")"; //最终转换为字符串形式
    setTimeout(changeColor,400);
}
```

```
var R = 0;
function change() {
    var temp = document.getElementsByTagName('body')[0];
    R=R+1;
    temp.style.backgroundColor = "rgb("+ R +",1,1)";
    setTimeout(change,10);
}
```

Window 下的 setTimeout(funcName, time)设置刷新

旋转的文字

利用 top 与 left 来定义一个对象的位置, 加一正弦余弦函数, 可实现旋转效果

```
<span id="round" style="position: absolute;left: 300px;top: 300px;color: red;font-size: 36px">无忧脚本</span>
<script type="text/javascript">
    var r=100; //半径
    var ins = 2; //圆的幅度
    function Circle() {
        var x = r*Math.cos(ins)+300;
        var y = r*Math.sin(ins)+300;
        ins+=0.02;
        var round = document.getElementById('round');
        round.style.left = x+"px";
        round.style.top = y+"px";
    }
    setInterval(Circle,5);
</script>
```

编辑控制

Style 属性的重要属性:

- readOnly = true / false 只读
- disabled = true /false 是否可编辑

改变样式-特别方法

```
<ul class="navul">
  <li onmouseover="id1.className='over'" onmouseout="id1.className='out'">公司产品</li>
  <ul id="id1" class="out">
    <li><a href="#">办公设备</a></li>
    <li><a href="#">会议设备</a></li>
  </ul>

```

???

事件处理

--事件驱动—以消息为基础，以事件为驱动

--浏览器事件机制：每发生一事件，总会生成一个事件对象 Event，DOM 元素把这个事件通知给浏览器，同时这个事件对象通常沿着 DOM 的树节点向上传播，直到文档顶部或者被脚本所捕获。

--事件：因某种交互行为发生导致文档需要做某些处理的场合。

--JS 中事件都以对应的属性表示，将一个闭包赋给这个属性，或者注册这个属性，该闭包即该事件的处理函数—句柄。

--事件响应：闭包响应一个事件时对应的 Event 对象总是被作为参数传递给这个闭包，闭包执行过程即事件响应。

浏览器事件驱动机制

- 简单事件模型—0 级 DOM 事件模型

通过简单赋值将时间句柄绑定给事件属性

- 标准时间模型—2 级 DOM 事件模型

- Internet Explorer 事件模型

- Netscape 事件模型

基本事件处理

HTML 标准事件类型

| 事件代理 | 事件说明 | 支持的 HTML 标记 |
|-------------|----------|--|
| Onabort | 图片装载被中断 | <object> |
| Onblur | 元素失去焦点 | <button><input><label><select><textarea><body> |
| onchange | 元素内容发生变化 | <input><select><textarea> |
| Onclick | 单击鼠标 | 大部分标记 |
| ondblclick | 双击鼠标 | 大部分标记 |
| Onerror | 图片装载失败 | <object> |
| Onfocus | 元素获得焦点 | <button><input><label><select><textarea><body> |
| onkeydown | 键盘被按下 | 表单元素和<body> |
| onkeypress | 键盘被按下并释放 | 表单元素和<body> |
| onkeyup | 键盘被释放 | 表单元素和<body> |
| Onload | 文档装载完毕 | <body><frameset><iframe><object> |
| onmousedown | 鼠标被按下 | 大部分标记 |
| onmousemove | 鼠标在元素上移动 | 大部分标记 |
| onmouseout | 鼠标移开了元素 | 大部分标记 |
| onmouseover | 鼠标移到元素上 | 大部分标记 |
| onmouseup | 鼠标被释放 | 大部分标记 |
| Onreset | 表单被重置 | <form> |
| onresize | 调整窗口大小 | <body><frameset><iframe> |
| onselect | 选中文本 | <input><textarea> |
| onsubmit | 表单被提交 | <form> |
| onunload | 写在文档或框架 | <body><frameset><iframe> |


```
var a = document.getElementById('div-1');
a.onabort = f(); // 图片转载中断
a.onblur = f(); // 元素失去焦点
a.onChange = f(); // 元素内容变化
a.onclick = f(); // 单击
a.ondblclick = f(); // 双击
a.onerror = f(); // 图片装载失败
a.onfocus = f(); // 获得焦点。
a.onkeydown = f(); // 键盘按下
a.onkeypress = f(); // 键盘按下并释放
a.onkeyup = f(); // 键盘释放
a.onload = f(); // 文档装载完毕
a.onmousedown = f(); // 鼠标被按下
a.onmousemove = f(); // 鼠标在元素上移动
a.onmouseout = f(); // 鼠标移开元素
a.onmouseover = f(); // 鼠标移到了元素上
a.onmouseup = f(); // 鼠标被释放
a.onreset = f(); // 表单被重置
a.onresize = f(); // 调整窗口大小
a.onselect = f(); // 选中文本
a.onSubmit = f(); // 表单被提交
a.onunload = f(); // 写在文档或框架
```

事件绑定

静态绑定：HTML 中直接使用 onclick 等

动态绑定：Js 中获取对象后赋值给 onclick 等

事件返回值

事件处理是异步机制，一般返回值无意义，但是一些返回值至关重要。

```
<!--onsubmit 返回false将会阻止提交; -->
<form action="http://www.baidu.com" onsubmit="return false;">
  <input type="submit">
</form>
```

标准事件模型

浏览器事件传播

事件传播分 3 阶段：

1. 捕捉阶段 capturing

事件从 Document 对象根节点沿着文档树向下到达目标节点，如果在这之间有注册了捕捉事件的函数，就执行。

2. 在目标节点上，运行注册的函数

3. 起泡阶段 bubbling

事件将从目标元素向上回传给 Document 根节点

注意：气泡阶段可能触发之间的句柄函数，所有不是所有的都适用于起泡

事件传播过程中任一阶段都能通过 Event 对象的 preventDefault() 来阻止默认动作发生。

事件注册—addEventListener()

addEventListener(para-1, para-2, para-3);

para-1: 参数 1, 注册的事件类型 (string), 如 "onclick"

para-2: 参数 2, 处理函数

para-3: 参数 3, Boolean, true—作用于事件传播的捕捉阶段

false—作用于目标自身和起泡阶段, 属常规

🕒 移除注册—removeEventListener() 参数与添加相同

关于 Event 接口

Event 对象必须实现 Event 接口或其子接口。

Event 接口是基础接口，UIEvent 和 MutationEvent 接口是其子接口，

MouseEvent 是 UIEvent 的子接口

Event 接口属性与方法

- 🕒 **Type**: 只读字符串，指明发生的事件类型。如“onclick”
- 🕒 **Target**: 只读，发生事件的节点
- 🕒 **EventPhase**: 只读，枚举类型，指定当前事件所处的事件传播阶段。三种

可能值:

1. `Event.CAPTURE_PHASE.`
2. `Event.AT_TARGET`
3. `Event.BUBBLING_PHASE`

- 🕒 **TimeStamp**: 只读，Date 类型，事件发生的时间戳
- 🕒 **Bubbles** : 只读，布尔，是否在文档中起泡
- 🕒 **Cancelable**: 只读，布尔，是否能否取消默认动作
- 🕒 **StopPropagation()**: 阻止当前事件从正在处理它的节点传播
- 🕒 **preventDefault()**: 阻止默认动作的执行

UIEvent 接口的属性

在父接口的基础上又定义了两个新的属性

- 🕒 **view**: 只读，发生事件的 window 对象
- 🕒 **detail**: 只读，提供事件的额外信息

MouseEvent 接口属性

在父接口 UIEvent 的基础上又定义了下列属性

- ☹ Button : 只读, 声明在 mouseDown/Up、click 事件中, 那个键改变了状态, 0: 左键; 2: 右键
- ☹ altKey / ctrlKey / metaKey / shiftKey 只读, 布尔, 声明了事件发生时是否按下了以上键;
- ☹ clientX / xlientY: 只读, 数值, 事件发生时鼠标相对于浏览器窗口的 X,Y 坐标
- ☹ screen X / screenY : 鼠标相对于屏幕的坐标
- ☹ relatedTarget: 只读, 引用与目标节点相关的节点 ; 对 nouseover, 它是移到目标上所离开的那个节点

JS与CSS

多种样式

```
<link rel="stylesheet" href="left.css" type="text/css" title="left"/>  
<link rel="alternate stylesheet" href="right.css" type="text/css" title="right"/>
```

在浏览器: 查看-页面格式

数据存储的脚本化

碍于浏览器安全限制, 数据存储一直是 JS 的短板

Cookie

- **Cookie** 是浏览器提供给客户端来存取少量数据的一种机制，**cookie** 是持久化对象，其生命周期长于 **window** 对象的周期。
- **Document** 对象的 **cookie** 属性是一个字符串，可以自身有许多属性，可以操作浏览器 **cookie** 对象。**Cookie** 对象是一个单例，在浏览器实例中是唯一的，不同窗口可以访问同一个 **cookie** 对象。

Cookie 属性

浏览器的 **cookie** 对象实际上可以看作一个关联数组，每个 **cookie** 成员有唯一的 **key: value** 对，

其他属性：

- 🕒 **Expires** 指定生存期。默认情况下 **cookie** 是暂存的，用户关闭浏览器后数据会被销毁。**Expires** 若设置一个失效日期，则数据不会销毁。

- 🌐 **Path** 指定与 **cookie** 关联在一起的网页。默认情况下会关联创建它的网页。设置为 **path = "/"** 时所有网页都关联，皆可访问。

Cookie 由于安全因素遵循 同源策略，因此在某个服务器上创建的 **cookie** 对象只在这个域才能访问。

- 🔒 **Secure** 布尔值，指定 **cookie** 传输方式，默认为 **false**——允许通过普通的/不安全的 **HTTP** 链接传输。设为 **true**——只能在 **https** 或其他安全协议下才能被传输。

- 🌐 **Domain**

Cookie 客服端的存取

Cookie 存储

格式:

```
document.cookie = "name1=value1;name2=value2...";  
document.cookie = "a = 100; b=200";
```

注意: cookie 值不能含有分号、逗号、空白符, 当有这些特殊字符出现时, 可以使用 js 提供的 `escape()` 函数进行编码, 但是读取时必须使用 `unescape()` 进行解码。

上述几个属性也可以添加到其中。

浏览器对 cookie 的限制及其严格, 规定浏览器保存的 cookie 数不超过 300, 每个 WEB 服务器保存的 cookie 数不超过 20, 每个数据不超过 4kb

Cookie 读取

```
var cookies = document.cookie.split(";");  
for (var i =0;i<cookies.length;i++){  
    var s = cookies[i].split("=");  
    if (s[0].trim()=="b"){  
        document.write(s[1].trim());  
    }  
}
```

Cookie 封装

```
function Cookie() {  
    this.set(name,value,expireTime){  
        if (!expireTime){  
            expireTime = new Date();  
        }  
        document.cookie =  
name+"="+value+"; "+"expire="+expireTime.toUTCString();  
    }  
}
```

```

    this.get = function (name) {
        var cookies = document.cookie.split(";");
        for (var i =0;i<cookies.length;i++){
            var s = cookies[i].split("=");
            if (s[0]==name){
                return s[1];
            }
        }
    }
}

var cookie = new Cookie();

```

userData

userData 是 IE 浏览器为 JS 提供的另一种数据存取手段

表 15.1 userData 与 cookie 的对比

| 比较标准 | Cookie | Userdata |
|-------|-------------|-------------------|
| 存储方式 | 字节流 | XML |
| 数据容量 | 小 (4KB* 20) | 64KB/页, 640KB/域 |
| 有效期 | 支持 | 支持 |
| 安全性 | 同源策略, 可修改子域 | 同源策略, 不允许修改子域 |
| 脚本兼容性 | 标准 | Internet Explorer |

同步与异步

注册的事件、定时器等都是异步执行的

setTimeout vs setInterval

- setTimeout()—超时调用

参数: 1. 字符串或函数

2. 时间

Js 是单线程解释器，一段时间只能执行一段代码，于是有一个任务队列，
setTimeout 的参数 2 说明多长时间将代码添加到队列中，如果队列是空的，
则立即执行，否则，添加的代码需要等待前面的任务完成后在执行。

返回值：返回一个 id，使用 id 能取消调用：

ClearTimeout(id)

- setInterval()—间歇调用

按照指定的时间间隔重复执行代码，参数同上

clearInterval

参数：字符串/闭包

字符串效率相对较低。

使用闭包的问题：在 setTimeout 调用时，实际的指针 this 也就是函数的所有者会发生变化，会出错，可再使用闭包纠正。

XML DOM 和 XML HTTP

XML DOM 和 XML HTTP 为 javascript 提供了读写 XML 文档的能力

(XML 比 HTML 内容更丰富，更复杂)

未学 XML，暂时略过

Ajax简介

Ajax:

A:Asynchronism

Ja: javascript

x: xml

未学，暂时略过

标准和兼容性

平台和浏览器的兼容性

最小公分母法

即避开使用不兼容的代码和实现

防御性编码

不能保证某段代码的正常运行时，添加额外代码来防御性地修复 BUG

Javascript 王者归来后面的超越 javascript 部分没有细读，还有 phototype

继承 多态等知识点没有细读。

localStorage 和 sessionStorage

HTML5 web 存储，一个比 cookie 更好的本地存储方式。

HTML5 Web 存储：数据不会被保存在服务器上，但是这些数据只用于用户请求

网站数据上。它也可以存储大量的数据，而不影响网站的性能。

- **LocalStorage：**没有时间限制

- `SessionStorage (window.sessionStorage)` : 针对一个 session, 关闭窗口、数据清空。

保存:

```
sessionStorage.setItem("key", "value");
```

取出:

```
var lastname = sessionStorage.getItem("key");
```

删:

```
sessionStorage.removeItem("key");
```

清空:

```
sessionStorage.clear();
```

注意事项

表单 radio 的 checked

单选框的 `checked` 属性是一个布尔属性, 存在即为选中, 不管是 `checked = 'checked'` 还是 `checked = true/false`

要使非选中，只有移除该属性。

Q: 如何根据单选框内容来实现其他形式（如 12/24 小时制切换）？

readOnly disable

可以通过标签属性设置，注意：disable = 'value' 或者 disable = "disable"

注册监听的方法

- 类 onclick = 方法是 DOM0 级规范，所有浏览器支持。但是不能绑定多个对象，使代码耦合。
- AddEventListener() 是 DOM2 规范，他可以控制在事件捕获阶段或者是冒泡阶段调用处理程序，但是只有 DOM2 的浏览器才支持（IE9+，FireFox，Safari、Chrome、opera）

Cursor

并不是只有 Button 才有 onclick, 使用 cursor 将 div 或其他元素，在焦点获取时，能够使用 onclick 等属性

| 值 | 描述 |
|-----------|---|
| url | 需被使用的自定义光标的URL 注释：请在此列表的末端始终定义一种普通的光标，以防没有由 URL 定义的可用光标。 |
| default | 默认光标（通常是一个箭头） |
| auto | 默认。浏览器设置的光标。 |
| crosshair | 光标呈现为十字线。 |
| pointer | 光标呈现为指示链接的指针（一只手） |
| move | 此光标指示某对象可被移动。 |
| e-resize | 此光标指示矩形框的边缘可被向右（东）移动。 |
| ne-resize | 此光标指示矩形框的边缘可被向上及向右移动（北/东）。 |
| nw-resize | 此光标指示矩形框的边缘可被向上及向左移动（北/西）。 |
| n-resize | 此光标指示矩形框的边缘可被向上（北）移动。 |
| se-resize | 此光标指示矩形框的边缘可被向下及向右移动（南/东）。 |
| sw-resize | 此光标指示矩形框的边缘可被向下及向左移动（南/西）。 |
| s-resize | 此光标指示矩形框的边缘可被向下移动（南/西）。 |
| w-resize | 此光标指示矩形框的边缘可被向左移动（西）。 |
| text | 此光标指示文本。 |
| wait | 此光标指示程序正忙（通常是一只表或沙漏）。 |
| help | 此光标指示可用的帮助（通常是一个问号或一个气球）。 |

添加属性与重置

对于一个自我给出的属性，不能通过 `.运算符` 来添加

```
main.style.character = value_;
```

而要使用 `name[a]= value` 格式：

```
main.style[character] = value_;
```

对于重置，其实十分简单，由于我们添加了 `style` 的诸多样式，于是我们移除样式 `style` 即可

```
reset.onclick = function () {
    main.removeAttribute('style');
}
```

闭包不能访问外围变量

```
for (let i=0; i<4; i++) {  
    let T = i; //注意，不同于Java，这里闭包是不能访问外围i的。  
    //必须使用另外一个值将外围的值保存起来，供闭包使用。  
    uls[i].onmouseover = function () {  
        for (let j = 0; j < 4; j++) {  
            ULS[j].style.display = "none";  
        }  
        ULS[T].style.display = "block";  
    }  
}
```

注意 let 的使用：

Let 是 ECMAScript6 中新增的，类似于 var，但是所声明的变量只在 let 命令所在的代码块内有效。

默认参数

为了防止函数调用 忘了参数，用 || 添加默认参数

```
function hello(txt){  
    txt = txt || 'hello world'  
}
```

ES6 更简单：

```
function hello(txt='hello world'){  
    ES6 格式  
}
```

兼容性问题

Opacity 透明度

支持 CSS3 的浏览器中，直接使用 `opacity=0` 即可

对于不支持 CSS3 的浏览器则要另寻它法

对 IE8 及以下浏览器，使用 `filter: alpha(opacity=0)`

(`filter` 是 ie 特有的)

如何从外部读取局部变量？

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

Js 代码

```
function f1(){  
    n=999;  
  
    function f2(){  
        alert(n); // 999  
    }  
  
}
```

