

JavaScript程序设计 (下)

2019.5.16

isszym sysu.edu.cn

目录

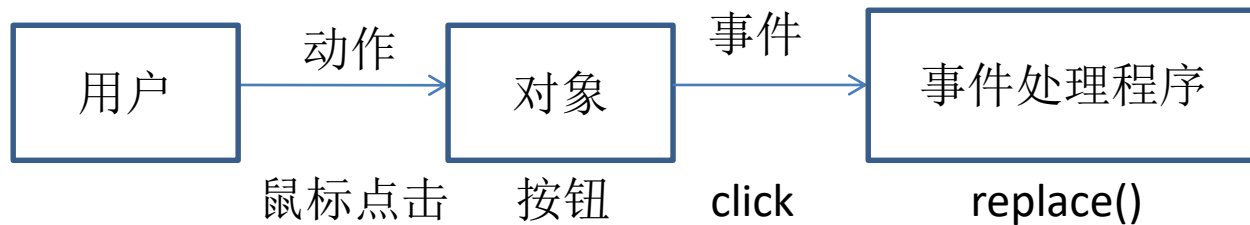
- + 事件绑定
- + 事件的捕捉阶段和冒泡阶段
- + 事件对象
- + 文档对象模型
- + AJAX技术
- + 定时器
- + BOM对象
- + 富文本编辑
- + Scroll与元素显示
- + 正则表达式
- + 性能问题

- + 附录1、DOM对象
- + 附录2、BOM对象
- + 附录3、其它对象
- + 附录4、富文本编辑框命令
- + 附录5、元素的其它操作
- + 附录6、DOM事件和方法
- + 附录7、事件对象(event)
- + 附录8、鼠标/键盘事件对象
- + 附录9、其它对象的方法
- + 附录10、IE8.0及更早版本的事件
- + 附录11、自定义属性的特征
- + 附录12、通过原型继承
- + 附录13、私有成员
- + 附录14、keycode编码表
- + 附录15、height、width与scroll
- + 附录16、获得元素的样式属性的值
- + 附录17、表单数据串行化
- + 附录18、jQuery的ready函数定义
- + 附录19、console对象
- + 附录20、控制台的调试命令
- + 附录21、JS编程规范
- + 附录22、MVVM
- + 附录23、ES6的部分新特性
- + 附录23、babel和webpack
- + 附录24、js常用框架
- + 附录25、参考资料

事件绑定

- 概述

- ✓ 当点击页面元素时会发生鼠标事件(event)，当输入字符时会产生按键事件，JavaScript可以捕捉到这些事件。
- ✓ 如果把事件和一段代码绑定(bind)在一起，可以实现在事件被触发时执行该段代码。与事件绑定的代码称为**事件处理程序**。



• 事件绑定方法

http://172.18.187.9:8080/js/

方法1：采用元素属性

onclick1.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Event binding</title>
  <script type = "text/javascript">
    function showMessage(){
      alert("Hello wolrd!");
    }
  </script>
</head>
<body>
  <input type="button" value="Click Me" onclick="alert(event.type)">
  <input type="button" value="Click Me" onclick="alert(this.value)">
  <input type="button" value="Hello" onclick="showMessage();">
  <input type="button" value="Click Me"
    onclick = "try {showMessage();}catch(ex){}">
</body>
</html>
```

显示Click

* `event.type` 取得事件类型Click, `this` 为被点击的对象

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
  function changetext(obj){
    obj.innerHTML="谢谢!";
  }
</script>
</head>
<body>
<div>
<p id="p1">请点击该文本</p>
<p id="p2">this is another paragraph</p>
</div>
<script type="text/javascript">
  var p1=document.getElementById("p1");
  var p2=document.getElementById("p2");
  var f1=function(){changetext(this)};
  p1.onclick= f1;
  p2.onclick= f1;
</script>
</body>
</html>
```

该语句要放在元素p1的后面。
函数外的语句会顺序会执行，
要求用到的对象已定义。

* 最好把希望立即执行的程序放在函数中执行，然后在网页尾部直接调用该函数。

方法2：采用addEventListener

eventlistener.html

```
var p1=document.getElementById("p1"); /* 执行要已定义元素p1 */  
var f1=function(){alert("hello");};  
p1.addEventListener("click",f1,false);/* false表示事件在冒泡阶段触发*/
```

addEventListener的参数：

第一个参数为要捕捉的事件

第二个参数为捕捉到事件后执行的函数

第三个参数false表示事件在冒泡阶段触发(默认)，为true则在捕捉阶段触发。

可以同时绑定多个事件处理程序：

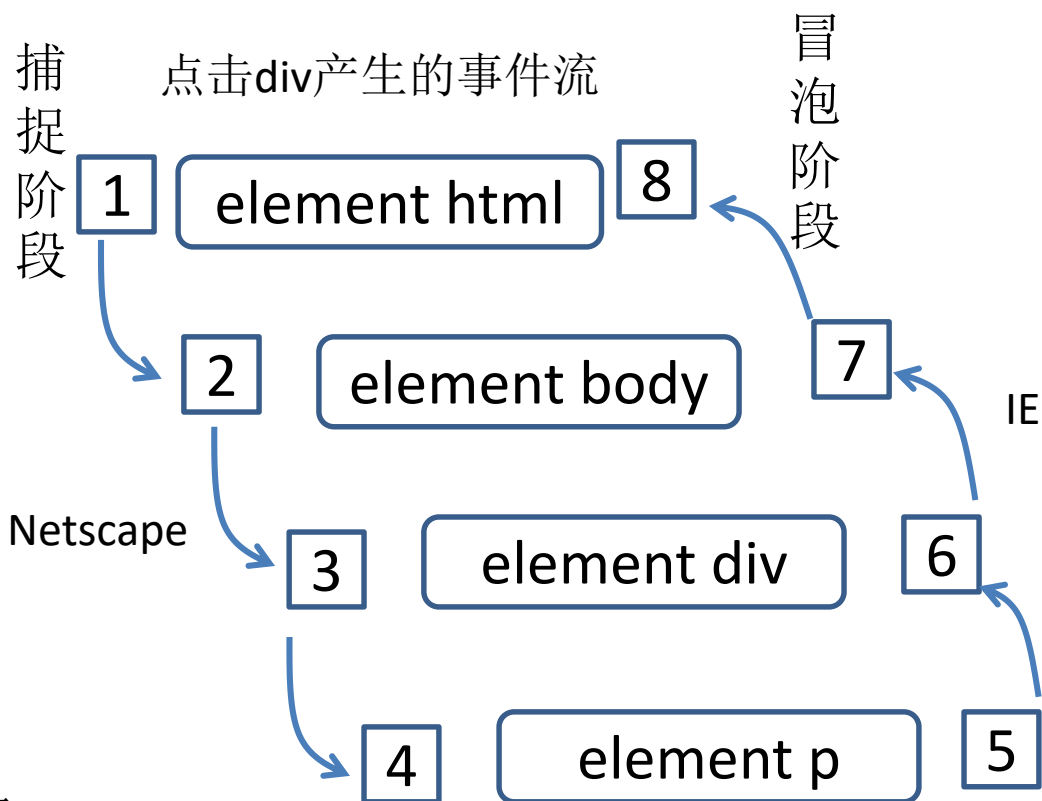
```
p1.addEventListener("click",function(){alert("Hello!");});  
p1.addEventListener("click",function(event){  
    alert(event.type+": "  
    +event.clientX+", "+event.clientY  
    +event.screenX+", "+event.screenY);  
});
```

事件的捕捉阶段和冒泡阶段

- 概念

```
<html id="html">
  <head>
    <title>events</title>
  </head>
  <body id="body">
    <h1>events</h1>
    <div id="div">
      <p id="p1">
        This is a paragraph.
      </p>
    </div>
  </body>
</html>
```

如果所有上面有id属性的元素都绑定click事件，点击p1时哪个先触发？



除了采用`addEventListener`绑定事件可以用于定义捕捉阶段和冒泡阶段触发事件，其它绑定事件的方法只在冒泡阶段触发事件。


```
<!DOCTYPE html>
<html id="html">
<head>
<meta charset="UTF-8">
<title>events</title>
</head>
<h1>addListener</h1>
<body id="body">
  <div id="div">
    <p id="p1">This is first paragraph.</p>
    <p id="p2">This is second paragraph.</p>
  </div>
</body>
<script>
  var p1 = document.getElementById("p1");
  var div = document.getElementById("div");
  var body = document.getElementById("body");
  var html = document.getElementById("html");
  html.addEventListener("click", function() {alert("html 捕捉");}, true);
  body.addEventListener("click", function() {alert("body 捕捉");}, true);
  div.addEventListener("click", function() {alert("div 捕捉");}, true);
  p1.addEventListener("click", function() {alert("p1 捕捉");}, true);
  p1.addEventListener("click", function() {alert("p1 冒泡");}, false);
  body.addEventListener("click", function() {alert("body 冒泡");}, false);
  div.addEventListener("click", function() {alert("div 冒泡");}, false);
  html.addEventListener("click", function() {alert("html 冒泡");}, false);
</script>
</html>
```

捕捉阶段

冒泡阶段

- 删除事件处理程序

```
btn.onclick = null;  
btn.removeEventListener("click",  
    function() {alert("Hello!");},  
    false);    //函数要完全一样才有效，否则，删除无效。
```

```
var handler = function(){alert(this.ad)};  
btn.addEventListener("click",handler, false);  
btn.removeEventListener("click",handler, false);  
    // 第三个参数与定义时也要一样
```

- 阻止事件传播和取消默认行为

//阻止该事件继续传播

```
var btn = document.getElementById("myBtn"); // myBtn为一个a元素的id  
btn.onclick = function(event){  
    event.stopPropagation();  
};
```

// 下面取消了链接的默认行为，即不会导航到url。 onclick="return false"效果类似

```
var link = document.getElementById("myLink");  
link.onclick = function(event){  
    event.preventDefault();    // 完整的例子见提交事件  
};
```

- 用程序产生事件

```
var p1=document.getElementById("p1");
var f1=function(){alert("hello");};
p1.addEventListener("click",f1,false);/* false表示事件在冒泡阶段触发*/
var ev = new MouseEvent('click', {
    cancelable: true,
    bubble: true,
    view: window });
p1.dispatchEvent(ev);
```

- 早期版本的IE

现在的Javascript基本统一了，IE 9.0 以及更早的版本有很多事件与现在的事件不同，具体情况见附录。

事件对象(event)

- 概述

[DOM事件](#)

- JS有用户界面事件、焦点事件、鼠标事件、键盘事件、触摸屏事件和手势事件六类事件。事件对象event是在触发某个事件时产生的对象，可以在事件处理程序中使用。
- event对象可以通过事件处理函数的参数得到:

```
function showEvent(evt){  
    alert(evt.type);  
}  
<input type="button" value="Click Me" onclick="showEvent(event)">
```

- event对象也可以在事件处理程序中直接得到:

```
function showEvent(){  
    var event=event||window.event||arguments.callee.caller.arguments[0];  
    alert(event.type);  
}  
<input type="button" value="Click Me" onclick="showEvent()">
```

* event赋值语句：window.event用于IE和Edge，最后一项用于Firefox，第一项用于其他浏览器

• 用户界面事件

load	页面或图像加载完毕	unload	退出页面
abort	加载中断	select	页面文本被选中
error	加载出错	resize	窗口或框架尺寸改变

图像、窗口和网页绑定加载事件：

- 图像装载完毕：``
- 网页全部对象装载完毕：`window.onload = function(){alert("Hello!")}`
`<body onload = "alert('Hello!')">`
- HTML文件装载完毕：`document.addEventListener('DOMContentLoaded', function(){alert(" DOM装载完毕")}, false);`
- 在文档状态变化事件发生时：
`document.onreadystatechange = function(){`
 `if(document.readyState=="complete")alert("全部Web对象装载完毕");`
 `if(document.readyState=="interactive")alert("html装载完毕");`
 `}`

* `window.onload`和`body.onload`为同一个事件，要在所有Web对象（html文件、css文件、js文件、image文件等）装载完毕后发生。

* 第三种是在html文件装载后立即发生，不用等所有Web对象装载完毕，与第四种的complete相同。

例子:

load.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>load</title>
    <script>
      document.onreadystatechange = function(){
        if(document.readyState=="complete")
          console.log('completely loaded');
        if(document.readyState=="interactive")
          console.log('html loaded');
      };
      document.addEventListener('DOMContentLoaded',
        function(){console.log('DOM loaded')});
      window.onload=function(){console.log('window loaded')};
    </script>
  </head>
  <body onload="console.log('body loaded')">
    
  </body>
</html>
```

执行结果: *html loaded*
DOM loaded
img loaded
completely loaded
body loaded

• 焦点事件

focus	输入域获得焦点事件
blur	输入域失去焦点事件
focusin	输入域通过冒泡方式得到焦点
focusout	输入域通过冒泡方式失去焦点

```
<input type= "text" onblur= "alert('blur')" onfocus="alert('focus')" >
```

例子:

focus.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>focus</title>
  </head>
  <body>
    <input onfocus="console.log('focus1')"
          onfocusin="console.log('focusin1')"
          onfocusout="console.log('focusout1')"
          onblur="console.log('blur1')">
    <input onfocus="console.log('focus2')"
          onblur="console.log('blur2')">
  </body>
</html>
```

进入第一个输入框时:

focus1
focusin1

进入第二个输入框时:

blur1
focusout1
focus2

再进入第一个输入框时:

blur2
focus1
focusin1

• 鼠标事件

click	单击
dblclick	双击
mousedown	按下鼠标
mouseup	放开鼠标
mouseenter	进入被选元素
mouseout	离开被选元素及其子元素
mousemove	在被选元素上方移动
mouseover	悬浮在被选元素上方

鼠标事件有继承性

```
<div onclick="alert('click')"> hello </div>
```

event.button	0	没按键
	1	按左键
	2	按右键
	3	按左右键
	4	按中间键
	5	按左键和中间键
	6	按右键和中间键
	7	按所有的键

鼠标点击的位置坐标

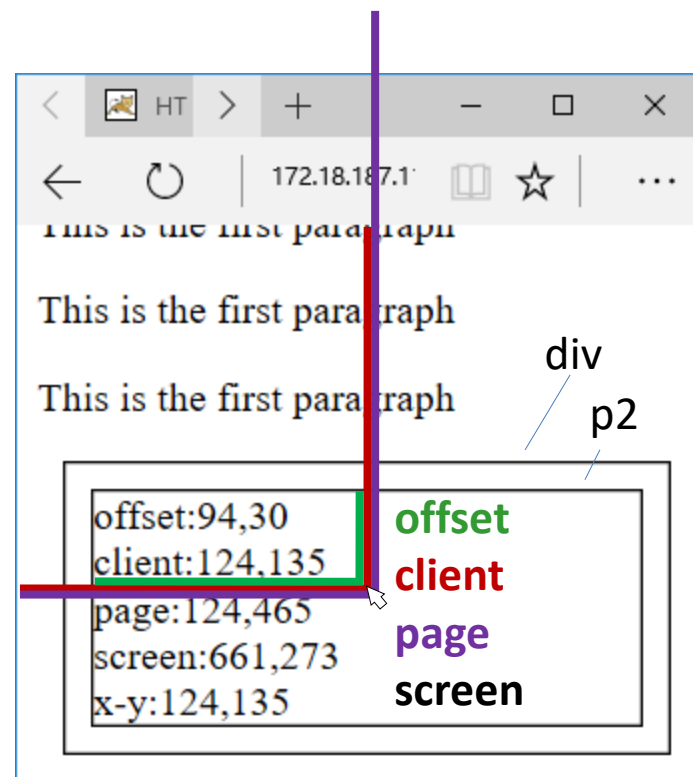
event.offsetX event.offsetY
event.clientX event.clientY
event.screenX event.screenY
event.pageX event.pageY
event.x event.y

* 父文档默认为BODY或relative元素

坐标参照元素

触发事件的元素
浏览器客户区
屏幕
当前页面
父文档进行定位

```
document.addEventListener('mousemove',  
    function(){  
        var x = "offset:" +event.offsetX  
                + event.offsetY;  
        var p2 = document.getElementById("p2");  
        p2.innerHTML=x; },  
    false);  
<p>This is the first paragraph!</p>  
<p id="p2">This is the second paragraph!</p>
```



`event.fromElement`

鼠标移动离开的对象

`event.toElement`

鼠标移动进入的对象

`event.srcElement`

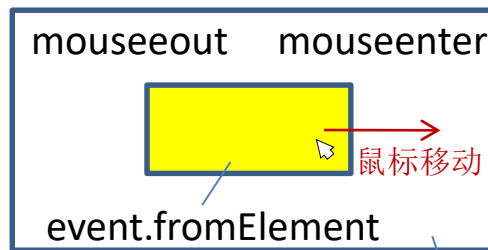
触发事件(鼠标点击)的元素

`event.target`

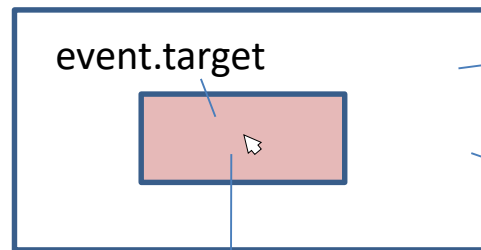
冒泡事件中的最小范围元素

`event.currentTarget`

设置当前事件的元素(同`this`)。



`event.toElement`



鼠标Click

绑定click事件

`event.currentTarget`

- * `mouseenter`和`mouseover`的`event.toElement`为当前元素(`this`)，`mouseout`的`event.fromElement`为当前元素(`this`)。
- * 当`document`绑定事件`moveout`时，离开哪个元素可以用`event.fromElement`获得。
- * 当`document`绑定事件`moveenter`时，进入哪个元素可以用`event.toElement`获得。

举例:

target.html

```
<!DOCTYPE html><html>
<body id="bd">
  <p id="p1">第一段</p><p id="p2">第二段</p><p id="p3">第三段</p><p id="msg"></p>
  <script>
    var msg = document.getElementById("msg");
    var p1 = document.getElementById("p1");
    var handler = function (event) {
      switch (event.type) {
        case "click": alert("Clicked"); break; //点击了myBtn之后显示Clicked
        case "mouseenter":
        case "mouseover":
          event.target.style.color = "red";
          msg.innerHTML="msg:" + event.toElement.id; break;
        case "mouseout":
          event.target.style.color = "";
          msg.innerHTML="msg:"+event.toElement.id;
      }
    };
    p1.onclick = handler;
    document.body.onclick = function (event) { //点击之后
      alert(event.currentTarget === document.body); //true
      alert(this === document.body); //true
      alert(event.target === event.currentTarget); //false
      alert(event.target === document.getElementById("p1")); //true (点击p1) 或false
    };
    p1.onmouseover = handler; //与后面四个一起选择其中一个进行测试
    //p1.onmouseenter = handler;
    //p1.onmouseout = handler;
    //document.onmouseover = handler;
    //document.onmouseout = handler;
  </script>
</body>
</html>
```

当点击p1, body的click事件:

event.currentTarget为本事件的实际定义者 (body)

event.target为本事件实际作用的元素 (p1)

- 键盘事件

[参考](#)

Keydown

按下任意键

Keyup

松开任意键

Keypress

按下字符键，长按则连续触发

`event.keyCode`

取到按键内码，即扫描码,详细见附录

`event.charCode`

取到按键的可显示字符

`event.altKey`

alt键是否按下，按下为**true**

`event.ctrlKey`

ctrl键是否按下，按下为**true**

`event.shiftKey`

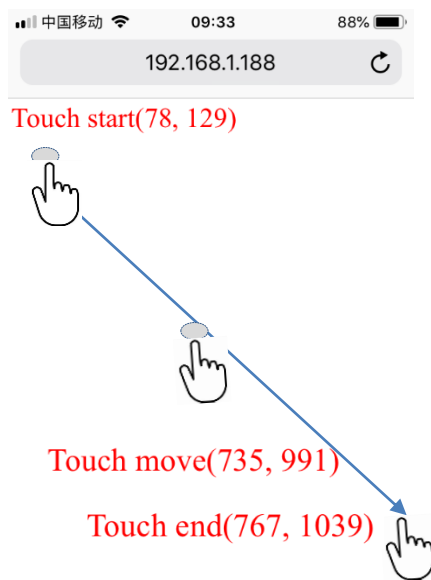
shift的键是否按下，按下为**true**

• 触摸屏事件

[参考](#) [参考](#) [参考](#)

```
var pos = "";
function touches(ev){
    var oDiv=document.getElementById('div1');
    if(ev.touches.length>1) return;// 多点触摸则返回
    if(ev.touches.length==1) // 单点触摸
        pos = '('+ ev.touches[0].clientX + ', '
            + ev.touches[0].clientY + ')';
    switch(ev.type){
        case 'touchstart': //手指触摸屏幕时触发
            oDiv.innerHTML='Touch start'+pos;
            ev.preventDefault();//阻止出现滚动条
            break;
        case 'touchmove': //手指在屏幕上移动时触发
            oDiv.innerHTML='Touch move'+pos; break;
        case 'touchcancel': //当系统停止跟踪触摸时触发
            oDiv.innerHTML='Touch cancel'+pos; break;
        case 'touchend': //手指从屏幕上离开时触发
            oDiv.innerHTML='Touch end'+pos; break;
    }
}
document.addEventListener('touchstart',touches,false);
document.addEventListener('touchend',touches,false);
document.addEventListener('touchmove',touches,false);
```

touches	当前跟踪的触摸操作的touch对象数组 (TouchList)
targetTouches	特定事件目标的touch对象数组
changeTouches	从上一次触摸以来发生了改变的touch对象数组



```

<!doctype html>
<html><head>
  <meta charset="utf-8">
  <title>touches</title>
  <style> body { font-size:60px; color:red;}</style>
  <script>
    var pos = "";
    function touches(ev){
      var oDiv=document.getElementById('div1');
      if(ev.touches.length>1) return //多手指触摸则返回
      if(ev.touches.length==1)      //单手指触摸
        pos = '('+ ev.touches[0].clientX+', '
              + ev.touches[0].clientY+')';
      switch(ev.type){
        case 'touchstart': //手指触摸屏幕时触发
          oDiv.innerHTML='Touch start'+pos;
          ev.preventDefault();//阻止出现滚动条
          break;
        case 'touchmove':  //手指在屏幕上移动时触发
          oDiv.innerHTML='Touch move'+pos; break;
        case 'touchcancel': //当系统停止跟踪触摸时触发
          oDiv.innerHTML='Touch cancel'+pos; break;
        case 'touchend':   //手指从屏幕上离开时触发
          oDiv.innerHTML='Touch end'+pos; break;
      }
    }
    document.addEventListener('touchstart',touches,false);
    document.addEventListener('touchend',touches,false);
    document.addEventListener('touchmove',touches,false);
  </script>
</head>
<body>
  <div id="div1">Touch Events</div>
</body></html>

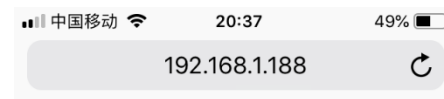
```

• 手势事件

在iOS中当两个手指操作的时候，就会产生手势，手势通常用于改变显示元素的大小，或者旋转显示的元素。**手势事件只能在iOS中运行，不能在安卓手机中使用。**手势事件分为三种：

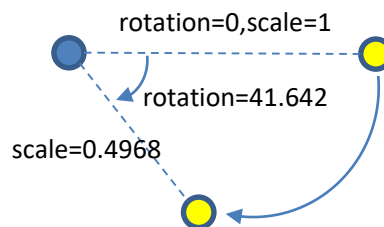
- gesturestart:** 当一个手指已经按在屏幕上，另一个手指又触摸屏幕的时候触发。类似于touchstart的作用一样。
- gesturechange:** 当触摸屏幕的任何一个手指的位置发生变化的时候触发。
- gestureend:** 当任何一个手指从屏幕上面移开时触发。

```
function gesture(ev) {  
    var div = document.getElementById('div1');  
    var ss='(rotation='+ev.rotation+', scale='+ev.scale+')';  
    switch (ev.type) {  
        case 'gesturestart':  
            div.innerHTML = 'Gesture start '+ss;  
            ev.preventDefault(); break;  
        case 'gestureend':  
            div.innerHTML = 'Gesture end ' + ss; break;  
        case 'gesturechange':  
            div.innerHTML = 'Gesture change ' + ss; break;  
    }  
}  
document.addEventListener('gesturestart', gesture, false);  
document.addEventListener('gestureend', gesture, false);  
document.addEventListener('gesturechange', gesture, false);
```



Gesture start (rotation=0, scale=1)

Gesture change
(rotation=41.64171600341797,
scale=0.4967755973339081)



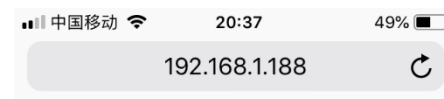
scale为相对于初始两点距离的缩放倍数
rotation是相对于初始两点线段的顺时针旋
转角度。反时针旋转为负值。




```

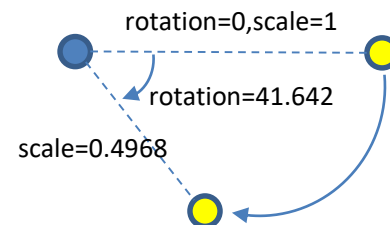
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>gesture</title>
  <meta name="viewport"
    content="width=device-width, minimum-scale=0.2,maximum-scale=0.2,user-scalable=no">
  <style> body { font-size: 20px;color: red;}</style>
  <script>
    window.onload = function() {
      function gesture(ev) {
        var div = document.getElementById('div1');
        var ss = '(rotation=' + ev.rotation + ', scale=' + ev.scale + ')';
        switch (ev.type) {
          case 'gesturestart':
            div.innerHTML = 'Gesture start '+ss;
            ev.preventDefault(); break;
          case 'gestureend':
            div.innerHTML = 'Gesture end ' + ss; break;
          case 'gesturechange':
            div.innerHTML = 'Gesture change ' + ss; break;
        }
      }
      document.addEventListener('gesturestart', gesture, false);
      document.addEventListener('gestureend', gesture, false);
      document.addEventListener('gesturechange', gesture, false);
    }
  </script>
</head>
<body>
  <div id="div1">gesture</div>
</body>
</html>

```



Gesture start (rotation=0, scale=1)

Gesture change
(rotation=41.64171600341797,
scale=0.4967755973339081)



scale 相对于初始两点距离的缩放倍数
rotation 相对于初始两点线段的顺时针旋转角度。反时针旋转为负值。

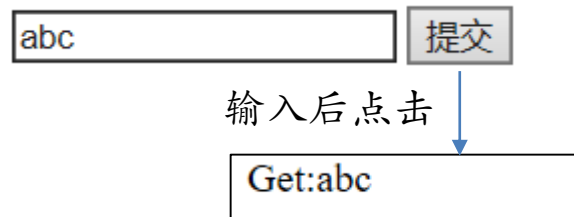
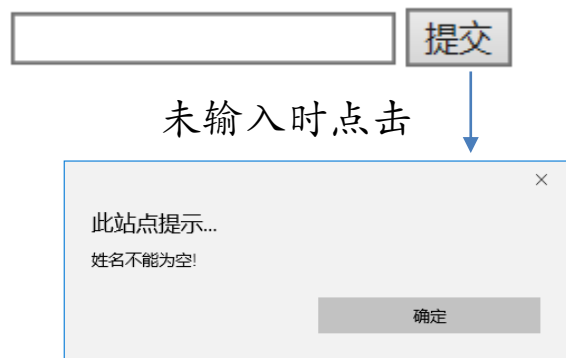


• 表单提交事件

如果希望在检查表单数据正确性之后再确定是否取消表单，就要有表单取消的方法。下面介绍几种表单取消方法：

方法1、通过点击submit按钮提交表单时，通过取消click事件可以取消提交操作。

```
<!DOCTYPE html><html>
<head> <meta charset="utf-8">
<script type="text/javascript">
    function submit1() {
        var p1 = document.getElementById("p1");
        if (p1.value == "") {
            alert("姓名不能为空!");
            return false;
        }
        return true;
    }
</script>
</head>
<body>
    <form name="frm1" method="post" action="post.jsp">
        <input id="p1" name="p1" type="text" />
        <input type="submit" onclick="return submit1();" value="提交" />
    </form>
</body></html>
```



* post.jsp见下页

方法2、点击submit按钮提交表单时，通过在事件处理程序中执行`event.preventDefault()`取消提交行为。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script type="text/javascript">
    function submit1(ev) {
      var p1 = document.getElementById("p1");
      if (p1.value == "") {
        alert("姓名不能为空!");
        ev.preventDefault();
        return;
      }
      return;
    }
  </script>
</head>
<body>
  <form name="frm1" method="post" action="post.jsp">
    <input id="p1" name="p1" type="text" />
    <input type="submit" onclick="submit1(event);" value="提交" />
  </form>
</body>
</html>
```

```
<%@ page language="java"
      import="java.util.*"
      contentType="text/html;"
      charset="utf-8"%>
<%request.setCharacterEncoding("utf-8");%>
<%
    String p1=request.getParameter("p1");
    out.print("Get:"+p1);
%>
post.jsp
```

方法3、如果采用button类型按钮，则点击它时不会引起表单提交，只有在事件处理程序中调用表单的submit方法才能提交。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script type="text/javascript">
    function submit1() {
      var p1 = document.getElementById("p1");
      if (p1.value == "") {
        alert("姓名不能为空!");
        return;
      }
      document.frm1.submit();
    }
  </script>
</head>
<body>
  <form name="frm1" method="post" action="post.jsp">
    <input id="p1" name="p1" type="text" />
    <input type="button" onclick="submit1();" value="提交" />
  </form>
</body>
</html>
```

* document.forms["frm1"]可以得到form

方法4、当提交表单时会产生的submit事件,如果事件处理程序返回true会完成提交行为,否则会取消提交行为。

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <script type="text/javascript">
    function submit1() {
      var p1 = document.getElementById("p1");
      if (p1.value == "") {
        alert("姓名不能为空!");
        return false;
      }
      return true;
    }
  </script>
</head>
<body>
  <form onsubmit = "return submit1()" method="post" action="post.jsp">
    <input id="p1" name="p1" type="text" />
    <input type="submit" value="提交" />
  </form>
</body>
</html>
```

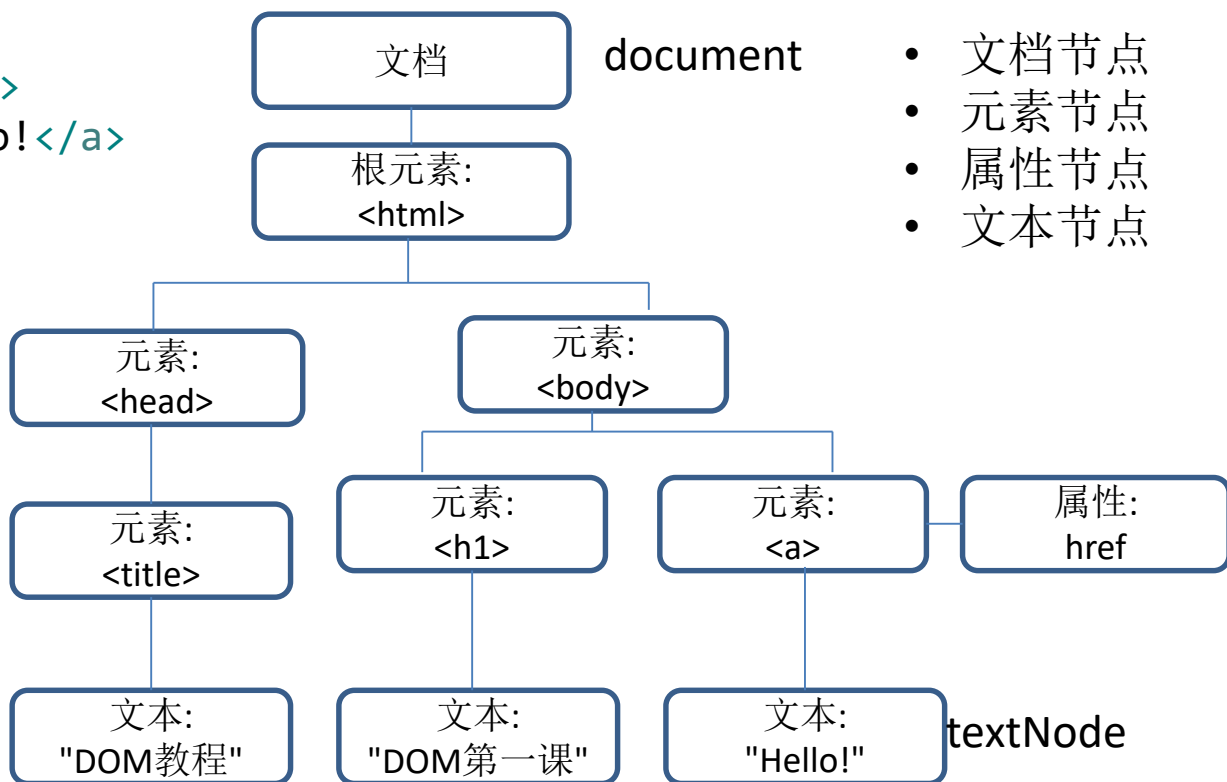
* 也可以采用event.preventDefault()取消submit事件,做法与方法2类似。

文档对象模型

(Document Object Model, DOM)

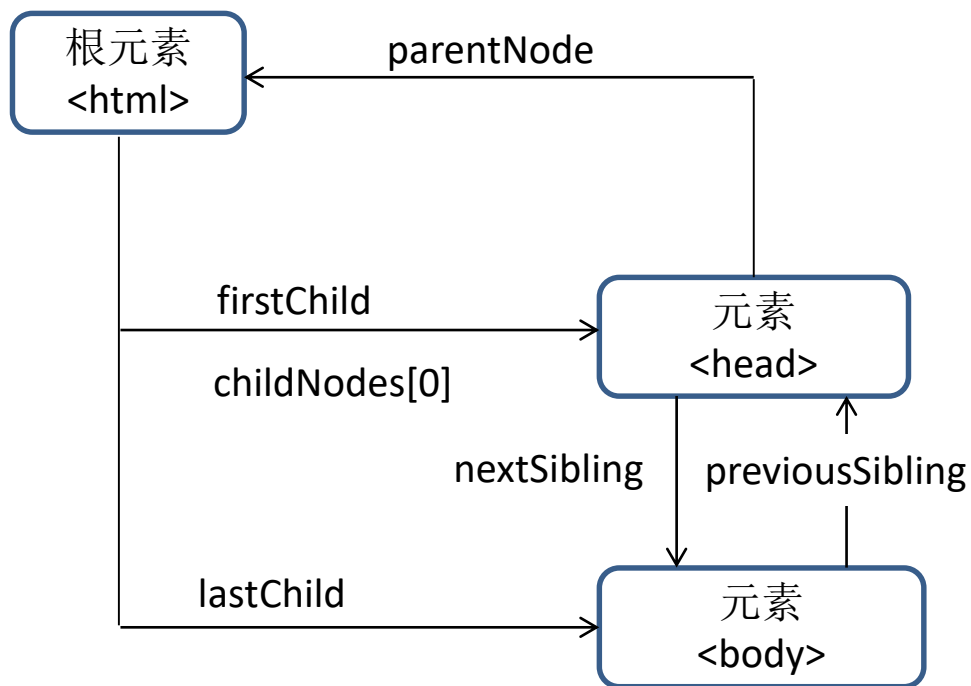
- 文档树

```
<html>
  <head>
    <title>DOM 教程</title>
  </head>
  <body>
    <h1>DOM 第一课</h1>
    <a href="#" >Hello!</a>
  </body>
</html>
```



- 文档节点
- 元素节点
- 属性节点
- 文本节点

* 每个节点都是一个对象



nodeName	标签名	元素节点
	属性名	属性节点
	#text	文本节点
	#document	文档节点
nodeType (节点类型)	1	元素
	2	属性
	3	文本
	8	注释
	9	文档

```

<div>
  <span id="s1">
    My text
  </span>
</div>

```

nodeValue (节点值)	文本	文本节点
	属性值	属性节点
	null	文档节点
	null	元素节点

```

var span = document.getElementById("s1");
alert(span.parentNode.nodeName);           //div

```


• 节点的其他属性

:

innerHTML 取到节点（元素）的内容。

outerHTML，取到本元素(包含内容)。

innerText 类似innerHTML，只是当成文本返回，只有IE支持。

outerText 类似outerHTML，只是当成文本返回，只有IE支持。

childNodes 会返回所有子节点（数组）。**children**与其类似，只是不会返回TextNode。

attributes 取到节点（元素）的所有属性

className 取得属性class的值。

classList 获得节点的所有类名。可以用以下方法修改它：

add(增加类)，**remove**（删除类）

toggle（切换类），**contains**（是否包含一个类）(HTML 5支持)

data- 自定义属性名（HTML5支持）

* 当元素内容为空格时，IE10之前的版本不会返回文本节点。为了统一所有浏览器的行为，html5为节点重新定义了一套遍历节点的属性：

firstElementChild，**lastElementChild**，**nextElementSibling**，**previousElementSibling**，**childElementCount**。

• 直接获得元素的方法

通过遍历的方法，从文档树的一个节点开始可以取到所有文档树的其他节点。除了这种方法，还可以通过**document**的一些方法直接得到所需要的节点。

<code>getElementById()</code>	得到具有某个id值的元素。
<code>getElementsByTagName()</code>	返回包含带有指定标签名称的所有元素的集合(nodelist)。
<code>getElementsByClassName()</code>	返回包含带有指定类名(可以是空格隔开的多个类名)的所有元素的集合。只有html5支持
<code>querySelectorAll()</code>	取得与CSS选择器匹配的所有元素的集合。只限html5
<code>querySelector()</code>	取得与CSS选择器匹配的第一个节点。只限html5

```
var s = document.querySelectorAll("a");//返回文档中所有的a元素的集合
document.write(s.length);
```

• 修改文档树的方法

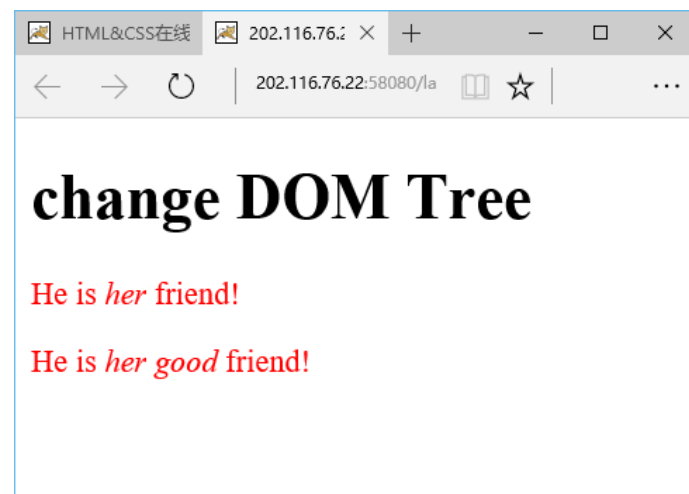
通过增删改子节点等方法可以直接改变文档树的结构，还可以修改节点属性（例如，`nodeValue`，`innerHTML`和`outerHTML`）或者通过修改元素属性和样式改变文档树。

例子1

```
<!DOCTYPE html><html><head>
<script type="text/javascript">
...//见下页
</script>
</head><body>
<h1>change DOM Tree</h1>
<div id="test"> </div>
</body></html>
```

增加或修改
节点后

```
<div id="test">
  <p id="para1" style="color:red">He is <em> her </em> friend! </p>
  <p style="color:red">He is <em> her good</em> friend! </p>
</div>
```

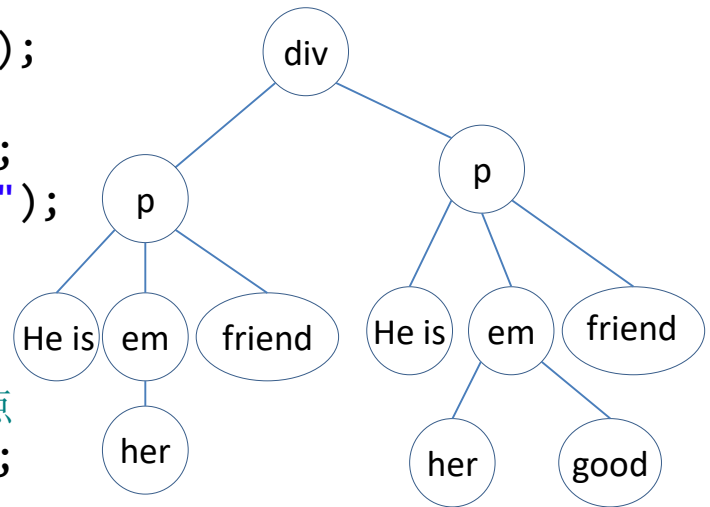


```

window.onload= function(){
    var para1 = document.createElement("p");
    var txt1 = document.createTextNode("He is ");
    var em1 = document.createElement("em");
    var txt2 = document.createTextNode(" her ");
    var txt3 = document.createTextNode("friend!");
    para1.appendChild(txt1);
    em1.appendChild(txt2);
    para1.appendChild(txt3);
    para1.insertBefore(em1,txt3); // 插入孩子节点
    var style=document.createAttribute("style");
    style.value="color:red";
    para1.setAttributeNode(style);
    var testdiv = document.getElementById("test");
    testdiv.appendChild(para1);
    var para2 = para1.cloneNode(true); // 参数: 是否克隆属性
    testdiv.insertBefore(para2,null);
    var txt4 = document.createTextNode(" ");
    para2.childNodes[1].appendChild(txt4);
    txt4.nodeValue = " good ";
    alert(para2.childNodes[1].childNodes[1].nodeValue);
    alert(testdiv.outerHTML);
}

```

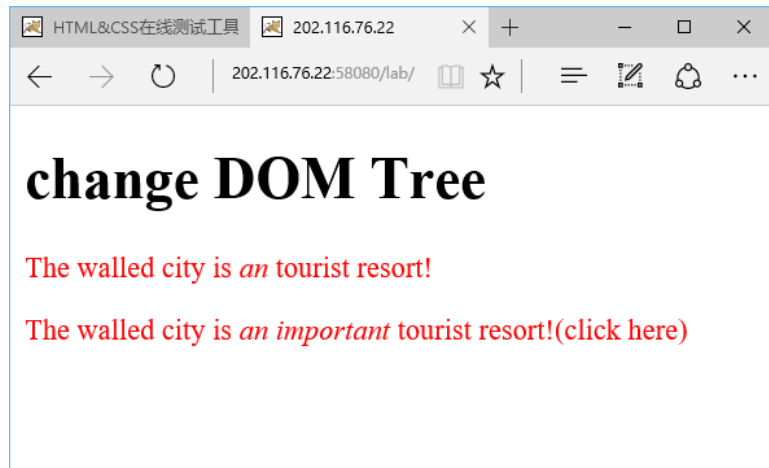
完成后的文档树



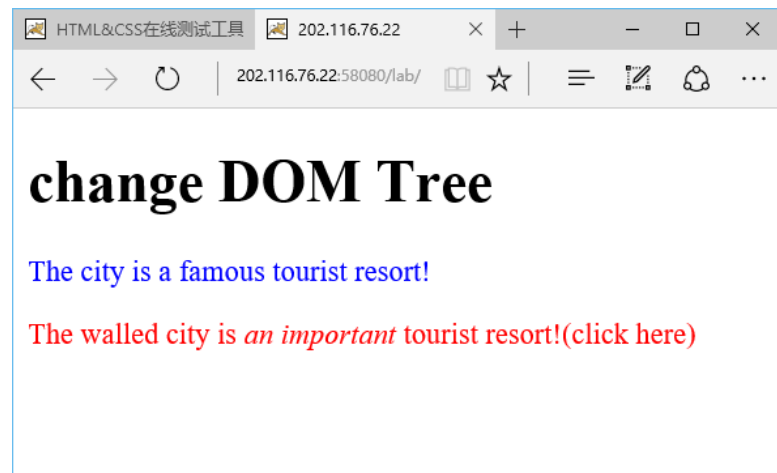
// 如果不采用window.load事件，而是直接执行，则需要把这段
 // Javascript代码放在body元素的最后面，即插入到在</body>前面。

例子2

```
<!DOCTYPE html><html><head>
<script type="text/javascript">
...
</script>
</head><body>
<h1>change DOM Tree</h1>
<div id="test">
  <p id="para1" style="color:red">
    The walled city is <em> an </em>tourist resort!
  </p>
  <p id="para2" style="color:red">
    The walled city is <em> an important </em>tourist resort!(click here)
  </p>
</div>
</body></html>
```



点击第
二行后
→



```

function insertAfter(newElement,targetElement){
    var parent = targetElement.parentNode;
    if(parent.lastChild==targetElement){
        parent.appendChild(newElement);
    }else{
        parent.insertBefore(newElement, targetElement.nextSibling);
    }
}

function removeNode(){
    var para1 = document.getElementById("para1");
    para1.removeChild(para1.childNodes[1]);
    para1.style.color = "blue";
    var txt1 = document.createTextNode(" a famous ");
    insertAfter(txt1,para1.childNodes[0]);
    var txt2 = document.createTextNode("The city is ");
    para1.replaceChild(txt2,para1.childNodes[0]);
}

window.onload= function(){
    var testdiv = document.getElementById("test");
    var para2 = document.getElementById("para2");

    testdiv.childNodes[3].setAttribute("onclick","removeNode()");
    //para2.onclick=function(){removeNode()}; //功能同上
    alert(testdiv.outerHTML);
}

```

因为HTML的元素很多都是对象，所以，除了采用DOM节点操作函数，还可以采用这些对象的特定方法修改DOM树：

```
<html><head><script type="text/javascript">
function insRow() {
    var x=document.getElementById('myTable').insertRow(0)
    var y=x.insertCell(0)
    var z=x.insertCell(1)
    y.innerHTML="NEW CELL1"
    z.innerHTML="NEW CELL2"
}
</script>
</head>
<body>
    <table id="myTable" border="1">
        <tr><td>Row1 cell1</td><td>Row1 cell2</td></tr>
        <tr><td>Row2 cell1</td><td>Row2 cell2</td></tr>
        <tr><td>Row3 cell1</td><td>Row3 cell2</td></tr>
    </table>
    <br />
    <input type="button" onclick="insRow()" value="insert row">
</body></html>
```

* 用Image img = new Image() 可以生成元素

* DOM对象见附录

• 获得和设置元素属性

通过属性节点的nodeValue可以获得或设置属性值，通过元素的方法getAttribute(name)和setAttribute(name,value)可以获得或设置属性值，还可以把属性名作为元素的数据域获得或修改属性值。

```
<div id="div1" class="hd" height="200px" width="400px" >
  <img id="logo" />
  <a id="a1" href="http://www.sysu.edu.cn">中大</a>
</div>
var div = document.getElementById("div1");
var img = document.getElementById("logo");
var a = document.getElementById("a1");
//取得元素属性
alert(div.id);
alert(div.getAttribute("id"));
alert(div.attributes["id"].nodeValue);
alert(div.className); //不用div.class是因为class是保留关键字
alert(div.getAttribute("class")); //注意这里是class而不是className

alert(div.height); //这里取到的是属性height，而不是style中的height
alert(div.width);
```

//设置元素特性

```
div.id = "div2";  
div.setAttribute("id", "div2");  
div.attributes["id"].nodeValue = "div2";  
img.src = "images/img1.gif";  
  
div.className = "ft";  
div.setAttribute("class", "ft");
```

//移除元素特性

```
div.removeAttribute("class");  
div.attributes.removeNamedItem("class");
```

```
a.href = "http://www.sina.com.cn";  
div.title = "header";  
a.innerHTML = "华工";
```

```
//重新设置超链接  
//title改为"header"  
// "中大"改为"华工"
```

• 修改元素样式

如果想修改元素的样式，可以通过修改样式文件名、修改style属性值、修改原子样式的值和修改类名来实现。文档中的每个样式表定义(外部css文件和style元素)都可以取到和修改。

```
<p id="hello" class="c1">Hello World!</p>
<link rel = "stylesheet" type="text/css" id="css" href="firefox.css" />
<!--改变样式文件名-->
<span onclick="javascript:document.getElementById('css').href = 'ie.css'">
    点我改变样式
</span>
<script>
    document.getElementById("hello").style.color="blue"; //修改原子样式
    document.getElementById("hello").className="c2"; //改变类属性值
    var p1 = document.getElementById('hello');
    p1.setAttribute("style", "color:blue"); //style属性
    p1.setAttribute("class", "c2 c3"); //改变类属性值
    // 取到第一个样式表（外部css文件、style元素）的内容并修改
    document.styleSheets[0].cssText =
        document.styleSheets[0].cssText.replace(/red/g, "yellow");
    // p1.style.cssText = "color:blue;width:800px";
</script>
```

AJAX技术

- 概述

传统的网页如果需要更新内容，必需重载整个网页。使用AJAX (Asynchronous Javascript And XML) 技术除了可以利用从 Web 服务器上获取的信息局部更新网页，还可以实现在保持当前网页的情况下提交数据给Web服务器。

2005年2月，Adaptive Path公司的Jesse James Garrett在文章“Ajax: A New Approach to Web Applications”中最早提出这个概念。这篇文章认为将XHTML、CSS、JavaScript、DOM和XMLHttpRequest混合使用来开发Web应用将会成为一种新的趋势。事实上，在Ajax这个概念出现之前就已经有了丰富的Ajax应用，例如，Google Maps和Google Suggest就是应用了XMLHttpRequest异步从服务器端来获取数据，实现了客户端无刷新的效果。

所有现代浏览器均支持 XMLHttpRequest 对象（IE5 和 IE6 使用 ActiveXObject）。

```
var xmlhttp;  
if (window.XMLHttpRequest) {  
    xmlhttp=new XMLHttpRequest();  
}else {  
    // code for IE6, IE5  
    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");  
}
```

[参考](#)

- 用iframe实现AJAX技术

更早期用到的异步技术是采用一个隐藏的iframe实现的。通过改变iframe的src来生成http请求，当收到http响应时会触发onload事件，此时可以取到http响应的正文。

```
<!DOCTYPE html><html><head>                                javascript10.jsp
<script type="text/javascript">
  function LoadContent(src, iframeID) {
    var content = document.getElementById(iframeID);
    content.setAttribute('src', src);
  }
  function swipContent(src, to) {
    var bobo=document.getElementById(src);
    var content = bobo.contentDocument?bobo.contentDocument.body.innerHTML
      :bobo.Document.body.innerHTML;
    document.getElementById(to).innerHTML = content;
  }
</script></head>
<body>
  <a href="#" onclick="LoadContent('ajaxGet.jsp','icontent')">Link Button</a>
  <iframe id="icontent" width="600" height="0" scrolling="auto" src=""
    frameborder="0" onload="swipContent('icontent','dest')"></iframe>
    <div id="dest" />
</body>
</html>
```

[参考](#) [参考](#)

• AJAX技术-GET

javascript7.jsp

```
function get() {  
    var xmlhttp = new XMLHttpRequest();           // 创建http请求  
    xmlhttp.onreadystatechange = function () {    // 当http请求的状态变化时执行  
        if (xmlhttp.readyState == 4) {          // 4-已收到http响应数据  
            if (xmlhttp.status >= 200 && xmlhttp.status < 300  
                || xmlhttp.status == 304) {      // 200~299-OK 304-unmodified  
                alert(xmlhttp.responseText);    // http响应的正文  
                var oTest = document.getElementById("p4");  
                oTest.innerHTML = xmlhttp.responseText;  
            } else {  
                alert("error");  
            }  
        }  
    };  
};  
// 打开http请求 (open) 的参数: get|post, url, 是否异步发送  
xmlhttp.open("get", "ajaxGet.jsp?p5=1&p6=5", true);  
xmlhttp.send(null);    //发送http请求。get只能用null作为参数  
}
```

* iframe和ajax只允许访问本网站的网页，不能进行跨网站访问。要允许网页被跨域访问，需要修改网页头部：response.addHeader("Access-Control-Allow-Origin", "*")。

```

<html><head>
  <script type="text/javascript">
    ...
  </script>
</head>
<body>
  <form name="p3">
    <p id="p4">hello world!</p>
    <input id="p1" type="text" value="Hello!" />
    <input id="p2" type="button" value="showHTML" />
  </form>
  <script type="text/javascript">
    p2 = document.getElementById("p2");
    p2.addEventListener("click", get);
  </script>
</body></html>

```

ajaxGet.jsp

```

<h1>DOM 1st Class </h1>
<p>Hello,world!</p>
<% request.setCharacterEncoding("utf-8");%>
<% String p5=request.getParameter("p5");
String p6=request.getParameter("p6");
out.print("Get: P5="+p5+",P6="+p6);
%>

```

hello world!

Hello!

showHTML

↑
点击

172.18.187.9:8080 显示

<h1>DOM 1st Class </h1>

<p>Hello,world!</p>

Get: P5=1, p6=5

确定

alert

• AJAX技术-POST

```
<script type="text/javascript">                                     javasrcipt8.jsp
    function post() {
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = function () {
            if (xmlhttp.readyState == 4) {
                if (xmlhttp.status >= 200 && xmlhttp.status < 300
                    || xmlhttp.status >= 304) {
                    alert(xmlhttp.responseText);
                    var oTest = document.getElementById("p4");
                    oTest.innerHTML = xmlhttp.responseText;
                } else {
                    alert("error");
                }
            }
        };
        var v1 = document.getElementById("p1").value;
        var param = "p5=" + encodeURIComponent(v1) + "&p6=345"; //汉字需要编码
        xmlhttp.open("post", "ajaxGet.jsp", true);
        xmlhttp.setRequestHeader("Content-Type",
                                "application/x-www-form-urlencoded");
        xmlhttp.send(param); // 没有参数就用null。
    }
</script>
```



```
<html><head>
  <script type="text/javascript">
    ...
  </script>
</head>
<body>
  <form name="p3">
    <p id="p4">hello world!</p>
    <input id="p1" type="text" value="Hello!" />
    <input id="p2" type="button" value="showHTML" />
  </form>
  <script type="text/javascript">
    p2 = document.getElementById("p2");
    p2.addEventListener("click", post);
  </script>
</body></html>
```

* 文件上传的AJAX技术见HTML5课件

• XMLHttpRequest对象

xmlhttp的属性:

responseText: 作为响应主体被返回的文本
responseXML: 如果响应的内容类型是"text/xml"或"application/xml",这里保存着响应数据。
status: 响应的HTTP状态码
statusText: HTTP状态说明。
readyState: 请求或响应过程的当前活动阶段。

xmlhttp.readyState

- 0: 未初始化。尚未调用open
- 1: 启动。已调用open, 未调用send
- 2: 发送。已调用send
- 3: 接收。已接收到部分响应数据
- 4: 完成。已接收到所有响应数据

xmlhttp.status

- 200~299: 表示成功接收请求, 并已经完成处理。
- 304: 请求的资源没有更改, 可以直接使用缓存数据。可以取到responseText。
- 其它: 有错或未完成请求。

设置http头部信息

```
xmlhttp.setRequestHeader("Content-Type",  
    "application/x-www-form-urlencoded");
```

设定超时时间

```
xmlhttp.timeout=1000; //ms  
xmlhttp.ontimeout=function(){  
    alert("no reponse received in a second.")  
};
```

定时器

语句`setTimeout(f, t)`用于在`t`毫秒之后执行一次函数`f`。如果每个`t`毫秒都要执行一次`f`，可以在函数`f`中执行`setTimeout(f, t)`。如果`setTimeout(f, t)`返回的值为`id`，`id`可以用于停止定时器的执行`clearTimeout(id)`。也可以使用`setInterval(f, t)`实现周期性执行函数`f`的功能，利用其返回值`id`也可以停止该定时器`clearInterval(id)`。

```
<html>
<head>
<script language="JavaScript" type="text/javascript">
    var i = 0;
    function hello() {
        i++;
        var ctrl = document.getElementById("timer");
        ctrl.innerHTML = i;
        window.setTimeout(hello, 1000);
    }
    var id=setTimeout(hello, 1000); //ms
</script>
</head>
<body>
    timer:<span id="timer">0</span>
</body>
</html>
```

timer.html

* 使用`setInterval(hello, 1000)`可能会因为在周期时间内没有执行完毕而出现异常，采用`setTimeout`实现周期执行功能就不会出现这种情况，但是`setTimeout`会耗费更多的管理时间。

BOM对象

(Browse Object Model)

- 概述

javascript可以通过一组对象获得浏览器属性或操作浏览器：

[参考](#)

- 通过window对象新建或访问当前浏览器窗口
- 通过document 对象访问每个载入浏览器的 HTML 文档（网页）
- 通过screen对象获得显示屏的特性
- 通过location对象得到和设置浏览器地址栏的内容
- 通过history对象操作浏览历史
- 通过navigator 对象获得浏览器本身的信息。
- 通过global对象可以使用全局变量和全局函数

Javascript的全局变量和函数都属于window对象。document、screen、location、history、navigator都属于window对象。对这些变量、函数和对象的访问可以省略window，例如，window.document可以写为document。

- window对象

window对象以用来确定浏览器窗口客户区大小，详细做法见附录。执行 `window.open("http://www.w3school.com.cn")` 可以打开一个窗口，还可以用参数来控制打开窗口的外观，例如，

```
window.open("http://www.w3school.com.cn", "_blank",  
            "toolbar=yes, location=yes, directories=no, status=no,  
            menubar=yes, scrollbars=yes, resizable=no,  
            copyhistory=yes, width=400, height=400")
```

- `_blank`表示在新窗口打开网页，`_self`表示在当前窗口打开网页，`name`表示在名称为`name`的窗口打开网页。
- `toolbar=yes, location=yes` 表示浏览器显示工具栏、地址栏
- `menubar=yes , scrollbars=yes`表示浏览器显示菜单和滚动条
- `directories=no, status=no, resizable=no`表示不显示收藏夹、状态栏，不可变化大小

window和body的load的事件是在整个页面的所有对象装载完毕后发生，如果希望html文件装载后立即发生则要使用document的事件DOMContentLoaded。

• location对象

通过location.href可以取到或设置URL地址，location.hash可以取到或者设置书签，通过location.hostname 和location.port还可以取到URL中的主机名和端口号。

• document对象

document对象可以取到当前网页的内容。document.anchors[]、document.forms[]和document.images[]可以得到当前网页的所有a元素、所有form和所有img元素（均为对象数组）。document.referrer可以得到载入当前网页的网页的URL，即从哪个网页(URL)点击进入当前网页的。document.cookie可以取到浏览器的cookie。**document.write(...)**可以直接往网页中输出一个字符串。事件DOMContentLoaded只要页面加载完毕就发生，而事件window.onload需要等待所有页面对象加载完毕才发生。采用document的方法可以获得元素，见前面“获取元素的方法”。

```
var s = document.querySelectorAll("a");  
document.write(s.length);
```

• cookie对象

下面的网页在加载之后立即检查是否存在username的cookie，如果存在，则取出其值，如果不存在，则要求输入值然后创建该cookie。

```
<!DOCTYPE html><html><head>
<script type="text/javascript">                                cookie.html
    function getCookie(c_name) {
        if (document.cookie.length > 0) {
            c_start = document.cookie.indexOf(c_name + "=");
            if (c_start != -1) {
                c_start = c_start + c_name.length + 1;
                c_end = document.cookie.indexOf(";", c_start);
                if (c_end == -1)
                    c_end = document.cookie.length;
                return unescape(document.cookie.substring(c_start, c_end));
            }
        }
        return "";
    }
    function setCookie(c_name, value, expiredays) {
        var exdate = new Date();
        exdate.setDate(exdate.getDate() + expiredays);
        document.cookie = c_name + "=" + escape(value) +
            ((expiredays == null) ? "" : ";expires="
            + exdate.toGMTString());
    }
}
```

escape()和unescape()为编码和解码函数，最好使用encodeURIComponent()和decodeURIComponent()代替。

```

function checkCookie() {
    var username = getCookie('username');
    if (username != null && username != "") {
        alert('Welcome again ' + username + '!')
    }
    else {
        username = prompt('Please enter your name:', "");
        if (username != null && username != "") {
            setCookie('username', username, 365)
        }
    }
}
</script>
</head>
<body onload="checkCookie()">
</body>
</html>

```

执行第一遍设置
执行第二遍显示

• history对象

执行history.back()、 history.forward()和history.go(-2)可以回退一页、前进一页和回退两页。

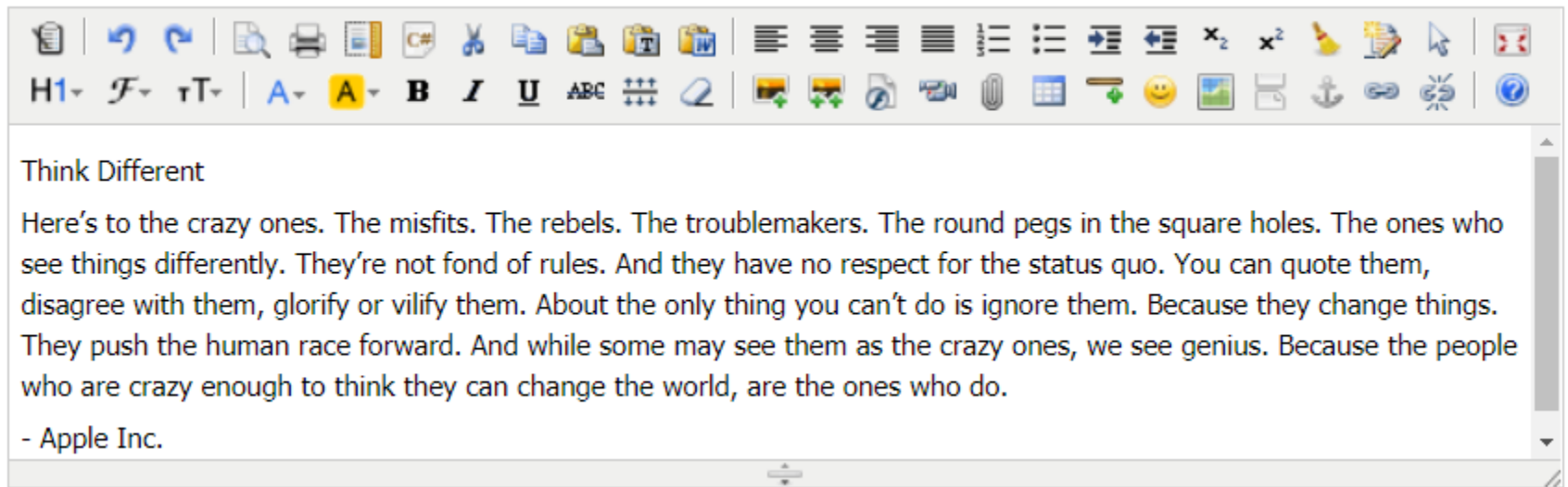
• 其它对象

日期操作采用Date对象，数学函数采用Math对象，数值转换使用Number对象， RegExp对象用于操作正则表达式，文档中很多元素都属于DOM对象，具体内容见附录。

富文本编辑

`input`元素或者`textarea`元素只能输入文字，富文本编辑可以在输入框中直接设置文本颜色、字体大小、显示图片等。

有两种实现富文本编辑的方法，一种是通过`iframe`实现，另一种是通过`div`实现。它们都可以对当前选中的内容通过执行命令设置样式。



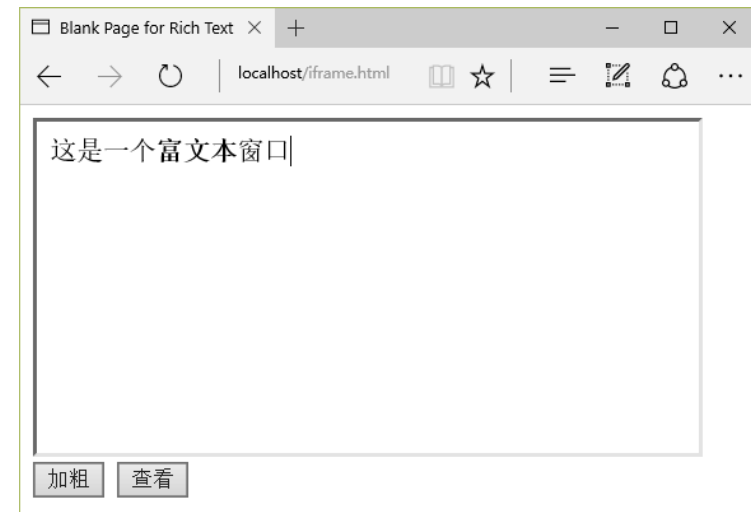
<http://kindeditor.net/demo.php>

```

<!DOCTYPE html>
<html><head> <title>Blank Page for Rich Text Editing</title></head>
<body>
  <iframe name="edit" style="height:200px;width:400px;">
  </iframe><br>
  <input type="button" value="加粗" onclick="bold()">
  <input type="button" value="代码" onclick="show()">
  <script>
    frames["edit"].document.designMode = "on";
    function bold() {
      frames["edit"].document.execCommand("bold", false, null);
    }
    function show() {
      var o = frames["edit"].document.body;
      alert(o.innerHTML);
    }
  </script>
</body>
</html>

```

选择一段文字，点击
按钮“加粗”

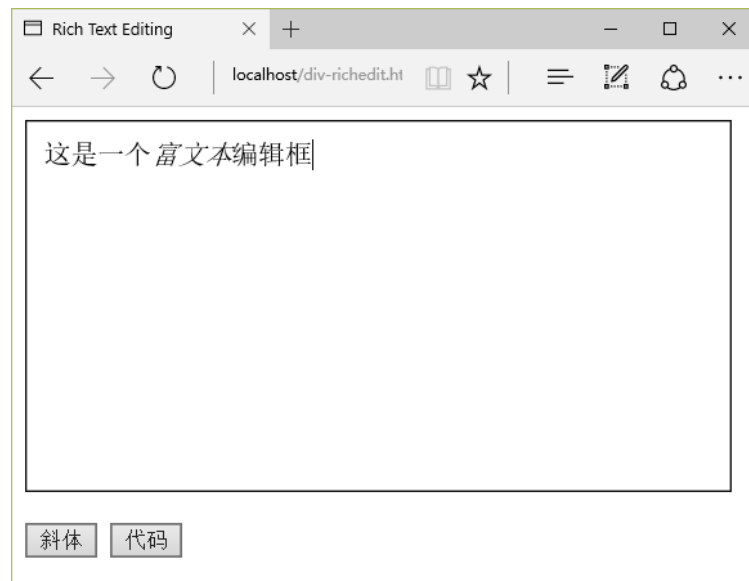


```

<!DOCTYPE html>
<html><head><title>Rich Text Editing</title></head>
<body>
  <div class="editable" id="richedit" contenteditable="true"
    style="padding:10px;width:400px;height:200px;border:solid 1px black">
  </div><br>
  <input type="button" value="斜体" onclick="italic()">
  <input type="button" value="代码" onclick="show()">
  <script>
    function italic() {
      document.execCommand("italic", false, null);
    }
    function show() {
      var o = document.getElementById("richedit");
      alert(o.innerHTML);
    }
  </script>
</body>
</html>

```

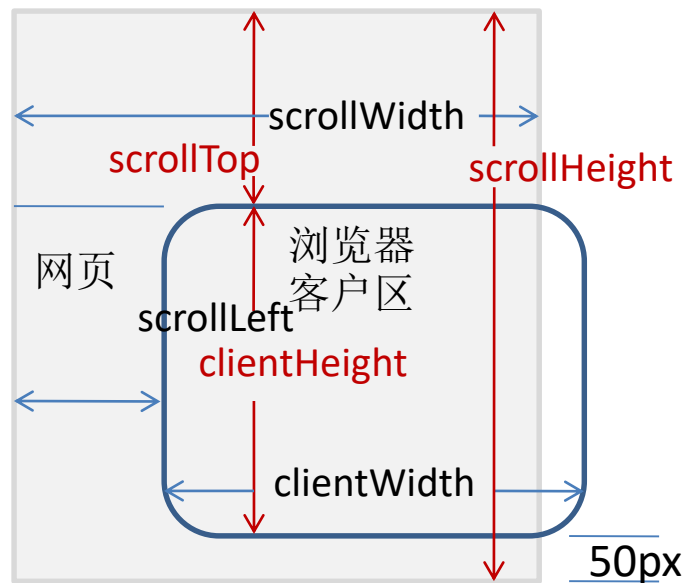
- * 所有元素都可以用于富文本编辑。
- * 获取选择的文档: `var selection = document.getSelection();`
- * `executeCommand`的其他命令见附录。



scroll与元素显示

- 如何判断网页已经滚动到页面底部？




```
function scrollTop() { //获取页面顶部被卷起来的高度
    return Math.max(
        document.body.scrollTop, // 除了Firefox Opera
        document.documentElement.scrollTop); //
}
function scrollHeight() { //获取页面的总高度
    return Math.max(document.body.scrollHeight,
        document.documentElement.scrollHeight);
}
function clientHeight() { //获取页面浏览器客户区的高度
    //document.compatMode=BackCompat: 标准兼容模式关闭; =CSS1Compat: 标准兼容模式开启。
    return (document.compatMode=="CSS1Compat")?document.documentElement.clientHeight
        :document.body.clientHeight;
}
```

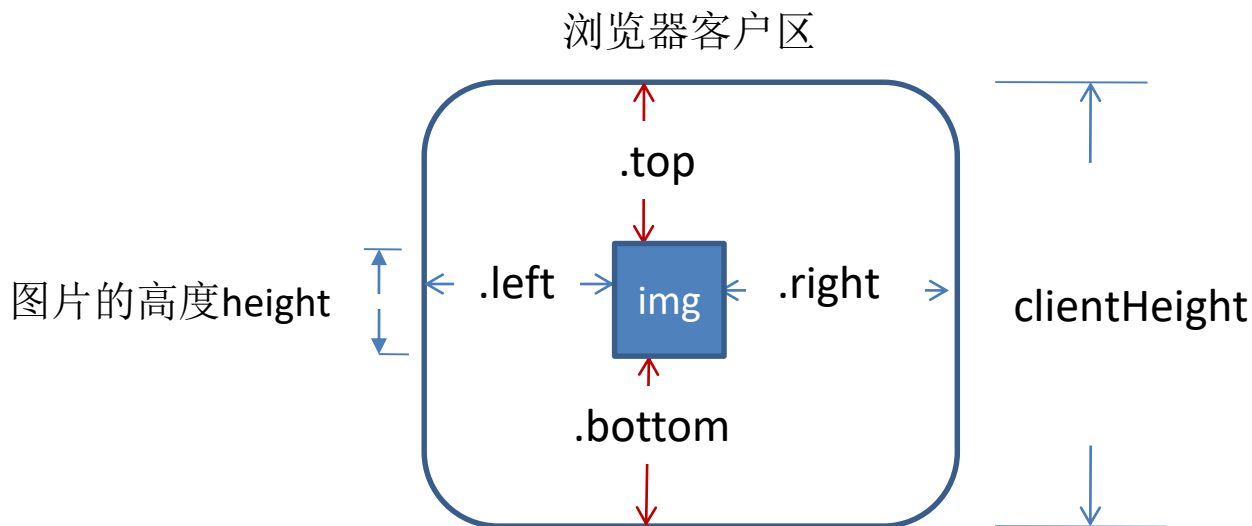


如果以下条件满足，说明网页已经滚动到(接近)页面底部：

$\text{scrollTop}() + \text{clientHeight}() \geq \text{scrollHeight}() - 50\text{px}$

- 如何判断一个图片是否出现在浏览器客户区？

一个图片对象的边框相对于浏览器客户区的位置可以采用.getBoundingClientRect()取到，.getBoundingClientRect().top是相对客户区上边沿的距离，可以取负值。



判断一个图片出现在浏览器客户区的方法是的上边框或下边框处于浏览器客户区：

```
.top > 0 && .top < clientHeight  
|| .bottom > 0 && .bottom < clientHeight
```

正则表达式

正则表达式可以用来在一个字符串中查找一个子串,，替换一个字符串等。

```
<!DOCTYPE HTML><html><head></head><body><script>
function prn(str) {
    document.write(str);
    document.write("<br>");
}
var sToMatch = "a bat, a Cat; a fAt caT"; // 要匹配的字符串
var reCat = new RegExp("at");           // 定义查找的子串at
prn(reCat.test(sToMatch));               // 返回true(找到子串).
prn(sToMatch.search(reCat));             // 返回3。如果未找到子串，则返回-1
reCat = new RegExp("at", "gi");          // 定义要查找或替换的子串：全部(g)，忽略大小写(i)
prn(sToMatch.replace(reCat, "ar"));      // 替换子串，返回a bar, a Car; a far car
prn(sToMatch.replace(reCat, function (sMatch) {
    if (sMatch == "At")                  // 子串At被替换为ast
        return "ast";
    else
        return "ar";                    // 其他子串，例如，at,aT，被替换为ar
}));                                     // 替换子串"At"和其它，返回a bar, a Car, a fast car
arrMatches = sToMatch.match(reCat);      // 匹配子串，返回一个数组at, at, At, aT
prn(arrMatches);
reCat = new RegExp(";|,");               // 用;或,划分子串
var arrWords = sToMatch.split(reCat);    // 返回一个数组: "a bat", "a Cat"和"a fAt caT"
prn(arrWords);
```

[参考](#)

```

// 除了用RegExp对象表示，正则表达式也可以用字面量表示：
prn(sToMatch.match(/cat/gi));          // 返回数组: "Cat","caT"
var reCat = /cat/gi;                    // 定义要查找或替换的子串：全部(g)，忽略大小写(i)
prn(sToMatch.replace(reCat, "dog"));    // a bat, a dog; a fAt dog
sToMatch = "ABCaACEFabCADDf";
var reCat = new RegExp("[A-Z]{3}", "g");
reCat.lastIndex = 0;
while ((result = reCat.exec(sToMatch)) != null) { //找出所有匹配
    var str = "str:" + result[0];
    var fIndex = reCat.firstIndex;
    var lIndex = reCat.lastIndex - result[0].length;
    str = str + "   Index:" + lIndex; //str:ABC Index:0   str:ACE Index:4 str:ADD Index:11
    prn(str);
}
// 以下为作为对照的普通字符串函数
prn(sToMatch.replace("at", "ar")); // 返回a bar, a Cat; a fAt caT。 替换第一个匹配子串
// (大小写敏感)。如果未找到子串，则返回-1
prn(sToMatch.search("aT")); // 返回21。
// 返回一个数组，包含两个子串:"a bat, a Cat"和"a fAt caT"
var arrSen = sToMatch.split(";");
</script>
</body>
</html>

```

```

<!DOCTYPE HTML><html> <head>    <title>reg1.html</title> </head>
<body><script>
    function prn(str) {
        document.write(str);
        document.write("<br>");
    }
    var sToMatch = "aaaabbbbbbbbaaabbbaabbbb123aaaaabbbb123"; // 要匹配的字符串
    var re1 = /. *bbb/g;      // 无限匹配, 尽可能往后匹配, .*包含尽可能多的字符
    var re2 = /. *?bbb/g;     // 加?表示懒惰匹配, 每次匹配到1个子串则返回, g表示要反复进行
    //var re3 = /. *+bbb/g;    // 无限匹配, 含义同re1, Javascript不可用
    prn(re1.test(sToMatch)); // true
    arrMatches = sToMatch.match(re1); // 返回匹配数组。无匹配项时返回null
    prn(arrMatches);           // aaaabbbbbbbbaaabbbaabbbb123aaaaabbbb
    arrMatches = sToMatch.match(re2);
    prn(arrMatches);          // aaaabbbb,bbb,baaabbbaabbbb,b123aaaaabbbb
    sToMatch = "dasdasblackredOUOI";
    prn(sToMatch.match(/(red|black|green)/g)); // black,red
    sToMatch = "Important word is the last one.";
    prn(sToMatch.match(/^(.+?)\b/g));         // Important  懒惰匹配, 找出第一个单词
    prn(sToMatch.match(/(\w+)\.$/g));         // one.   (找出最后一个单词)
    prn(sToMatch.match(/\b(\S+?)\b/g));       // Important,word,is,the,last,one  找出所有单词
    prn(sToMatch.match(/(\w+)/g));            // 结果同上
    sToMatch = "This is a string using badword1 and anotherbadword.";
    prn(sToMatch.replace(/badword1|anotherbadword/g, "*****"));
                                // This is a string using **** and ****
    prn(sToMatch.replace(/badword1|anotherbadword/g,
        function (sMatch) { return sMatch.replace(/./g, "*"); }));
                                // This is a string using ***** and *****

```



```

sToMatch = "#1234567890";
prn(sToMatch.match(/#\d+/g));           // #1234567890 未捕捉
prn(RegExp.$1);                          // ""
prn(sToMatch.match(/#(\d+)/g));          // #1234567890 捕捉
prn(RegExp.$1);                          // 1234567890
sToMatch = "1234 5678";
prn(sToMatch.replace(/(\d{4}) (\d{4})/g, "$2 $1")); // 5678 1234
prn(RegExp.$1);                          // 1234
prn(RegExp.$2);                          // 5678
sToMatch = "dogdog";
prn(sToMatch.match(/(dog)\1/));           // dogdog,dog \1代表第一个反向引用
prn(RegExp.$1);                          // dog
prn(RegExp.$2);                          // ""
sToMatch = "#1234567890";
prn(sToMatch.match(/#(?:\d+)/g));         // #1234567890 取消捕捉(?:)
prn(RegExp.$1);                          // ""
sToMatch = "<b>this would be bold</b>";
prn(sToMatch.replace(/<(?:.|\s)*?>/g, "")); // this would be bold 取消捕捉,同/<?:.*?>/
prn("<br>==多行模式:/m==");
sToMatch = "First second\nthird fourth\nfifth sixth";
prn(sToMatch.match(/(\w+)$/g));           // sixth
prn(sToMatch.match(/(\w+)$/gm));          // second,fourth,sixth 每一行最后词
prn("<br>==前瞻==");
var sToMatch1 = "bedroom";
var sToMatch2 = "bedding";

```

```

prn(sToMatch1.match(/(\bed(?:=room))/)); //前瞻。IE不支持
prn(sToMatch2.match(/(\bed(?:=room))/)); //前瞻。IE不支持
prn(sToMatch1.match(/(\bed(?:!room))/)); //后瞻。IE不支持
prn(sToMatch2.match(/(\bed(?:!room))/)); //后瞻。IE不支持
prn(RegExp.$1);
prn("<br>==RegExp==");
sToMatch = "bbq is a short for barbecue";
var reB = /b/g;
prn(reB.global); // true. g
prn(reB.ignoreCase); // false. i
prn(reB.multiline); // false. m
prn(reB.source); // b
reB.exec(sToMatch);
prn(reB.lastIndex); // 1
reB.exec(sToMatch);
prn(reB.lastIndex); // 2
reB.exec(sToMatch);
prn(reB.lastIndex); // 20
reB.exec(sToMatch);
prn(reB.lastIndex); // 23
reB.exec(sToMatch);
prn(reB.lastIndex); // 0
re1 = /(s)hort/g;
re1.test(sToMatch);
prn(RegExp.input); // "bbq is a short for barbecue"要匹配的字符串 简写为RegExp["$ _"]
prn(RegExp.leftContext); // "bbq is a " 匹配子串左边的字串 简写为RegExp["$ `"]
prn(RegExp.rightContext); // " for barbecue" 匹配子串右边的字串 简写为RegExp["$ '"]
prn(RegExp.lastMatch); // "short" 最后匹配的字符串 简写为RegExp["$ &"]
prn(RegExp.lastParen); // "s" 最后匹配的分组 简写为RegExp["$ +"]
</script></body> </html>

```

规则定义:

[...] 位于括号之内的任意字符
[^...] 不在括号之中的任意字符
. 除了换行符之外的任意字符,等价于`[^\n]`
`\w` 任何单字字符,等价于`[a-zA-Z0-9]`
`\W` 任何非单字字符,等价于`[^a-zA-Z0-9]`
`\s` 任何空白符,等价于`[\t\n\r\f\v]`
`\S` 任何非空白符,等价于`[^\t\n\r\f\v]`
`\d` 任何数字,等价于`[0-9]`
`\D` 除了数字之外的任何字符,等价于`[^0-9]`
`[\b]` 一个退格直接量(特例)

`{n, m}` 匹配前一项至少n次,但是不能超过m次
`{n, }` 匹配前一项n次,或者多次
`{n}` 匹配前一项恰好n次
? 匹配前一项0次或1次,等价于`{0, 1}`,即可选的。
+ 匹配前一项1次或多次,等价于`{1, }`
* 匹配前一项0次或多次,等价于`{0, }`

| 选择.匹配的要么是该符号左边的子表达式,要么它右边的子表达式
(...) 分组.将几个项目分为一个单元.这个单元中可使用*、+、? 和|等符号,而且还可以记住和这个组匹配的字符以供此后引用使用
`\n` 和第n个分组所匹配的字符相匹配.分组是括号中的子表达式(可能是嵌套的).分组号是从左到右计数的左括号数
^ 匹配的是字符的开头,在多行检索中,匹配的是一行的开头
\$ 匹配的是字符的结尾,在多行检索中,匹配的是一行的结尾
`\b` 匹配的是一个词语的边界.简而言之就是位于字符`\w` 和 `\w`之间的位置(注意:`[\b]`匹配的是退格符)
`\B` 匹配的是非词语的边界的字符
i 执行大小写不敏感的匹配
g 执行一个全局的匹配,简而言之,就是找到所有的匹配,而不是在找到第一个之后就停止了

* RegExp采用引号时的斜杠\要用双斜杠\\, 引号采用\", 例如`RegExp("\\(\\w\\)\\","g")`找出所有的"(字母数字串)"模式。

\f	换页符
\n	换行符
\r	回车
\t	制表符
\v	垂直制表符
\/	一个 / 直接量
\\	一个 \ 直接量
\.	一个 . 直接量
*	一个 * 直接量
\+	一个 + 直接量
\?	一个 ? 直接量
\	一个 直接量
\(一个 (直接量
\)	一个) 直接量
\[一个 [直接量
\]	一个] 直接量
\{	一个 { 直接量
\}	一个 } 直接量
\XXX	由十进制数 XXX 指定的ASCII码字符
\Xnn	由十六进制数 nn 指定的ASCII码字符
\cX	控制字符^X. 例如, \cl等价于
\t, \cJ	等价于 \n

贪婪重复：采用*和+，只要符合匹配模式就会尽可能多地包含字符。
非贪婪重复：在量词*和+后加上?，表示最小匹配，即匹配尽可能少的字符。

```
var content="abc37&*&defdasabc48*&n+_+def";
var reCat = new RegExp(/abc(.|\n)+?def/g);
var all = "", ind=0;
reCat.lastIndex = 0;
while ((result = reCat.exec(content)) != null) { //匹配两次
    var res = "" + result[0];
    var str = "str:" + res;
    var lIndex = reCat.lastIndex - res.length;
    str = str + " Index:" + lIndex;
    all = all + content.substring(ind,lIndex) + " " + res;
    document.write(str);
}
```

//去掉正则表达式的?只会匹配一次，res包含整个字符串

[replace](#)

性能问题

- 闭包效率低

由于闭包需要占用额外空间，而且使用了闭包的對象不能被銷毀，跨区域执行还会带来性能损失，一般最好避免出现闭包。解决方法就是尽量不使用外部引用，例如，采用参数带入所需要的引用。

- 减慢网页显示速度

JavaScript程序可以分成多个script元素，并可以放在html的任何地方。在默认情况下，浏览器顺序加载和解析或执行所有页面元素，下载执行外部文件往往会减慢网页显示速度。

带有src的script元素可以用属性defer和async来克服这个问题，async等待js文件异步下载完才执行它，defer还会等待加载和解析所有页面元素完成后再执行。另一种解决方法是把script元素放在页面的末尾（</body>之前）。

```
<script src="main.js" defer> </script>
```

- 单线程问题

JavaScript默认采用单线程执行，随着大量的界面处理和数据处理都搬到了前台，会导致JavaScript程序运行时间太长，使浏览器出现未响应警告。HTML5引入Web Worker的多线程解决方案克服这个问题。具体参见HTML5的课件。

附录

- [附录1、DOM对象](#)
- [附录2、BOM对象](#)
- [附录3、其它对象](#)
- [附录4、富文本编辑框命令](#)
- [附录5、元素的其它操作](#)
- [附录6、DOM事件和方法](#)
- [附录7、事件对象\(event\)](#)
- [附录8、鼠标/键盘事件对象](#)
- [附录9、其它对象的方法](#)
- [附录10、IE8.0及更早版本的事件](#)
- [附录11、自定义属性的特征](#)
- [附录12、通过原型继承](#)
- [附录13、私有成员](#)
- [附录14、keycode编码表](#)
- [附录15、height、width与scroll](#)
- [附录16、获得元素的样式属性的值](#)
- [附录17、表单数据串行化](#)
- [附录18、jQuery的ready函数定义](#)
- [附录19、console对象](#)
- [附录20、控制台的调试命令](#)
- [附录21、JS编程规范](#)
- [附录22、MVVM](#)
- [附录23、babel和webpack](#)
- [附录24、js常用框架](#)
- [附录25、参考资料](#)

附录1、DOM对象

[Document](#)

[Anchor](#)

[Area](#)

[Base](#)

[Body](#)

[Button](#)

[Canvas](#)

[Event](#)

[Form](#)

[Frame](#)

[Frameset](#)

[IFrame](#)

[Image](#)

[Input Button](#)

[Input Checkbox](#)

[Input File](#)

[Input Hidden](#)

[Input Password](#)

[Input Radio](#)

[Input Reset](#)

[Input Submit](#)

[Input Text](#)

[Link](#)

[Meta](#)

[Object](#)

[Option](#)

[Select](#)

[Style](#)

[Table](#)

[TableCell](#)

[TableRow](#)

[Textarea](#)

```
var img1 = new Image(); // HTML5 Constructor
img1.src = 'image1.png';
img1.alt = 'alt';
document.body.appendChild(img1);
var img2 = document.createElement('img');
img2.src = 'image2.jpg';
img2.alt = 'alt text';
document.body.appendChild(img2);
// using first image in the document
alert(document.images[0].src);
```

[参考1](#) [参考2](#)

*TextArea要用.value改变值，不能采用InnerHTML？

附录2、BOM对象

location 对象

- location 对象用于获得当前页面的地址 (URL), 并把浏览器重定向到新的页面。通过location.href可以返回当前页面的url, 设置它可以转到 相应页面。

Location 对象属性

属性	描述
hash	设置或返回从井号 (#) 开始的 URL (锚)。
host	设置或返回主机名和当前 URL 的端口号。
hostname	设置或返回当前 URL 的主机名。
href	设置或返回完整的 URL。
pathname	设置或返回当前 URL 的路径部分。
port	设置或返回当前 URL 的端口号。
protocol	设置或返回当前 URL 的协议。
search	设置或返回从问号 (?) 开始的 URL (查询部分)。

Location 对象方法

属性	描述
assign()	加载新的文档。
reload()	重新加载当前文档。
replace()	用新的文档替换当前文档。

history对象

- 通过history 对象可以访问浏览器历史。

History 对象属性

属性

[length](#)

描述

返回浏览器历史列表中的 URL 数量。

History 对象方法

方法

[back\(\)](#)

[forward\(\)](#)

[go\(\)](#)

描述

加载 history 列表中的前一个 URL。

加载 history 列表中的下一个 URL。

加载 history 列表中的某个具体页面。history.go(-1)与 history.back()作用相同。

window对象

window 对象表示浏览器窗口。所有 JavaScript 全局对象、函数以及变量均自动成为 window 对象的成员。全局变量是 window 对象的属性。全局函数是 window 对象的方法。HTML DOM 的 document 也是 window 对象的属性之一。

确定浏览器窗口客户区的尺寸（不包括工具栏和滚动条）。

```
var w=window.innerWidth || document.documentElement.clientWidth ||  
    document.body.clientWidth;  
var h=window.innerHeight || document.documentElement.clientHeight ||  
    document.body.clientHeight;
```

Internet Explorer 8、7、6、5

对浏览器窗口的操作：

- window.open() - 打开新窗口
- window.close() - 关闭当前窗口
- window.moveTo() - 移动当前窗口
- window.resizeTo() - 调整当前窗口的尺寸

属性	描述
closed	返回窗口是否已被关闭。
defaultStatus	设置或返回窗口状态栏中的默认文本。
document	对 Document 对象的只读引用。请参阅 Document 对象 。
history	对 History 对象的只读引用。请参阅 History 对象 。
innerheight	返回窗口的文档显示区的高度。
innerwidth	返回窗口的文档显示区的宽度。
location	用于窗口或框架的 Location 对象。请参阅 Location 对象 。
Navigator	对 Navigator 对象的只读引用。请参阅 Navigator 对象 。
outerheight	返回窗口的外部高度。
outerwidth	返回窗口的外部宽度。
Screen	对 Screen 对象的只读引用。请参阅 Screen 对象 。
self	返回对当前窗口的引用。等价于 Window 属性。
status	设置窗口状态栏的文本。
top	返回最顶层的先辈窗口。

事件: onload

方法	描述
alert()	显示带有一段消息和一个确认按钮的警告框。
blur()	把键盘焦点从顶层窗口移开。
clearInterval()	取消由 setInterval() 设置的 timeout 。
clearTimeout()	取消由 setTimeout() 方法设置的 timeout 。
close()	关闭浏览器窗口。
confirm()	显示带有一段消息以及确认按钮和取消按钮的对话框。
createPopup()	创建一个 pop-up 窗口。
focus()	把键盘焦点给予一个窗口。
moveBy()	可相对窗口的当前坐标把它移动指定的像素。
moveTo()	把窗口的左上角移动到一个指定的坐标。
open()	打开一个新的浏览器窗口或查找一个已命名的窗口。
print()	打印当前窗口的内容。
prompt()	显示可提示用户输入的对话框。
resizeBy()	按照指定的像素调整窗口的大小。
resizeTo()	把窗口的大小调整到指定的宽度和高度。
scrollBy()	按照指定的像素值来滚动内容。
scrollTo()	把内容滚动到指定的坐标。
setInterval()	按照指定的周期（以毫秒计）来调用函数或计算表达式。
setTimeout()	在指定的毫秒数后调用函数或计算表达式。

document对象

每个载入浏览器的 HTML 文档都会成为 Document 对象。
Document 对象使我们可以从脚本中对 HTML 页面中的所有元素进行访问。

document 对象的集合

集合	描述	IE	F	O	W3C	属性	描述	IE	F	O	W3C
all[]	提供对文档中所有 HTML 元素的访问。	4	1	9	No	body	提供对 <body> 元素的直接访问。				
anchors[]	返回对文档中所有 Anchor 对象的引用。	4	1	9	Yes		对于定义了框架集文档，该属性引用最外层的 <frameset>。				
applets	返回对文档中所有 Applet 对象的引用。	-	-	-	-	cookie	设置或返回与当前文档有关的所有 cookie。	4	1	9	Yes
forms[]	返回对文档中所有 Form 对象引用。	4	1	9	Yes	domain	返回当前文档的域名。	4	1	9	Yes
images[]	返回对文档中所有 Image 对象引用。	4	1	9	Yes	lastModified	返回文档被最后修改的日期和时间。	4	1	No	No
links[]	返回对文档中所有 Area 和 Link 对象引用。	4	1	9	Yes	referrer	返回载入当前文档的文档的 URL。	4	1	9	Yes
						title	返回当前文档的标题。	4	1	9	Yes
						URL	返回当前文档的 URL。	4	1	9	Yes

document 对象的方法

方法	描述
close()	关闭用 <code>document.open()</code> 方法打开的输出流，并显示选定的数据。
getElementById()	返回对拥有指定 id 的第一个对象的引用。
getElementsByName()	返回带有指定名称的对象集合。
getElementsByTagName()	返回带有指定标签名的对象集合。
getElementsByClassName()	返回带有指定类名的对象集合
open()	打开一个流，以收集来自任何 <code>document.write()</code> 或 <code>document.writeln()</code> 方法的输出。
write()	向文档写 HTML 表达式 或 JavaScript 代码。
writeln()	等同于 <code>write()</code> 方法，不同的是在每个表达式之后写一个换行符。

事件：

`window.onload`和`document.ready`的不同：一个要所有页面对象加载完毕加载完毕才发生，一个只要页面加载完毕就发生。
<http://www.cnblogs.com/a546558309/p/3478344.html>

html5

属性	描述
<code>readyState</code>	取值"loading"和"complete"，表示正在加载文档和文档加载完毕。
<code>head</code>	返回文档的head元素。 <code>document.body</code> 返回body元素。
<code>charset</code>	文档中实际使用的字符集
<code>documentElement</code>	把document当成元素对待

scrollIntoView()

方法	描述
<code>querySelectorAll()</code>	返回与CSS选择符匹配的所有节点。
<code>querySelector()</code>	返回与CSS选择符匹配的第一个节点。
<code>matchSelector()</code>	判断元素与CSS选择符是否匹配。很多浏览器不支持
<code>hasFocus()</code>	判断文档是否获得了焦点

screen对象

Screen 对象包含有关客户端显示屏幕的信息。

Screen 对象属性

属性	描述
availHeight	返回显示屏幕的高度 (除 Windows 任务栏之外)。
availWidth	返回显示屏幕的宽度 (除 Windows 任务栏之外)。
bufferDepth	设置或返回调色板的比特深度。
colorDepth	返回目标设备或缓冲器上的调色板的比特深度。
deviceXDPI	返回显示屏幕的每英寸水平点数。
deviceYDPI	返回显示屏幕的每英寸垂直点数。
fontSmoothingEnabled	返回用户是否在显示控制面板中启用了字体平滑。
height	返回显示屏幕的高度。
logicalXDPI	返回显示屏幕每英寸的水平方向的常规点数。
logicalYDPI	返回显示屏幕每英寸的垂直方向的常规点数。
pixelDepth	返回显示屏幕的颜色分辨率（比特每像素）。
updateInterval	设置或返回屏幕的刷新率。
width	返回显示器屏幕的宽度。

Navigator 对象

Navigator 对象包含有关浏览器的信息。

Navigator 对象属性

属性	描述
appName	返回浏览器的代码名。
appMinorVersion	返回浏览器的次级版本。
appName	返回浏览器的名称。
appVersion	返回浏览器的平台和版本信息。
browserLanguage	返回当前浏览器的语言。
cookieEnabled	返回指明浏览器中是否启用 cookie 的布尔值。
cpuClass	返回浏览器系统的 CPU 等级。
onLine	返回指明系统是否处于脱机模式的布尔值。
platform	返回运行浏览器的操作系统平台。
systemLanguage	返回 OS 使用的默认语言。
userAgent	返回由客户机发送服务器的 user-agent 头部的值。
userLanguage	返回 OS 的自然语言设置。

Navigator 对象方法

方法	描述
javaEnabled()	规定浏览器是否启用 Java 。
taintEnabled()	规定浏览器是否启用数据污点 (data tainting)。

Global对象(window)

函数

[decodeURI\(\)](#)

[decodeURIComponent\(\)](#)

[encodeURI\(\)](#)

[encodeURIComponent\(\)](#)

[escape\(\)](#)

[eval\(\)](#)

[getClass\(\)](#)

[isFinite\(\)](#)

[isNaN\(\)](#)

[Number\(\)](#)

[parseFloat\(\)](#)

[parseInt\(\)](#)

[String\(\)](#)

[unescape\(\)](#)

描述

解码某个编码的 URI。

解码一个编码的 URI 组件。

把字符串编码为 URI。

把字符串编码为 URI 组件。

对字符串进行编码。

计算 JavaScript 字符串，并把它作为脚本代码来执行。

返回一个 JavaScript 的 JavaClass。

检查某个值是否为有穷大的数。

检查某个值是否是数字。

把对象的值转换为数字。

解析一个字符串并返回一个浮点数。

解析一个字符串并返回一个整数。

把对象的值转换为字符串。

对由 `escape()` 编码的字符串进行解码。

属性

[Infinity](#)

[java](#)

[NaN](#)

[Packages](#)

[undefined](#)

描述

代表正的无穷大的数值。

代表 `java.*` 包层级的一个 `JavaPackage`。

指示某个值是不是数字值。

根 `JavaPackage` 对象。

指示未定义的值。

[js urlencode , encodeURIComponent](#)

附录3、其它对象

- Date对象
- Math对象
- cookie对象
- Number对象
- String对象
- RegExp对象
- DOM对象

Date对象

```
var now=new Date()  
document.write(now.getMonth())
```

Date 对象方法

方法	描述
Date()	返回当日的日期和时间。
getDate()	从 Date 对象返回一个月中的某一天 (1 ~ 31)。
getDay()	从 Date 对象返回一周中的某一天 (0 ~ 6)。
getMonth()	从 Date 对象返回月份 (0 ~ 11)。
getFullYear()	从 Date 对象以四位数字返回年份。
getYear()	请使用 getFullYear() 方法代替。
getHours()	返回 Date 对象的小时 (0 ~ 23)。
getMinutes()	返回 Date 对象的分钟 (0 ~ 59)。
getSeconds()	返回 Date 对象的秒数 (0 ~ 59)。
getMilliseconds()	返回 Date 对象的毫秒(0 ~ 999)。
getTime()	返回 1970 年 1 月 1 日至今的毫秒数。
getTimezoneOffset()	返回本地时间与格林威治标准时间 (GMT) 的分钟差。
getUTCDate()	根据世界时从 Date 对象返回月中的一天 (1 ~ 31)。
getUTCDay()	根据世界时从 Date 对象返回周中的一天 (0 ~ 6)。
getUTCMonth()	根据世界时从 Date 对象返回月份 (0 ~ 11)。
getUTCFullYear()	根据世界时从 Date 对象返回四位数的年份。
getUTCHours()	根据世界时返回 Date 对象的小时 (0 ~ 23)。
getUTCMinutes()	根据世界时返回 Date 对象的分钟 (0 ~ 59)。
getUTCSeconds()	根据世界时返回 Date 对象的秒钟 (0 ~ 59)。
getUTCMilliseconds()	根据世界时返回 Date 对象的毫秒(0 ~ 999)。
parse()	返回1970年1月1日午夜到指定日期（字符串）的毫秒数。

方法	描述
setDate()	设置 Date 对象中月的某一天 (1 ~ 31)。
setMonth()	设置 Date 对象中月份 (0 ~ 11)。
setFullYear()	设置 Date 对象中的年份（四位数字）。
setYear()	请使用 setFullYear() 方法代替。
setHours()	设置 Date 对象中的小时 (0 ~ 23)。
setMinutes()	设置 Date 对象中的分钟 (0 ~ 59)。
setSeconds()	设置 Date 对象中的秒钟 (0 ~ 59)。
setMilliseconds()	设置 Date 对象中的毫秒 (0 ~ 999)。
setTime()	以毫秒设置 Date 对象。
setUTCDate()	根据世界时设置 Date 对象中月份的一天 (1 ~ 31)。
setUTCMonth()	根据世界时设置 Date 对象中的月份 (0 ~ 11)。
setUTCFullYear()	根据世界时设置 Date 对象中的年份（四位数字）。
setUTCHours()	根据世界时设置 Date 对象中的小时 (0 ~ 23)。
setUTCMinutes()	根据世界时设置 Date 对象中的分钟 (0 ~ 59)。
setUTCSeconds()	根据世界时设置 Date 对象中的秒钟 (0 ~ 59)。
setUTCMilliseconds()	根据世界时设置 Date 对象中的毫秒 (0 ~ 999)。
toSource()	返回该对象的源代码。
toString()	把 Date 对象转换为字符串。
toTimeString()	把 Date 对象的时间部分转换为字符串。
toDateString()	把 Date 对象的日期部分转换为字符串。
toGMTString()	请使用 toUTCString() 方法代替。
toUTCString()	根据世界时，把 Date 对象转换为字符串。
toLocaleString()	根据本地时间格式，把 Date 对象转换为字符串。
toLocaleTimeString()	根据本地时间格式，把 Date 对象的时间部分转换为字符串。
toLocaleDateString()	根据本地时间格式，把 Date 对象的日期部分转换为字符串。
UTC()	根据世界时返回 1970 年 1 月 1 日到指定日期的毫秒数。
valueOf()	返回 Date 对象的原始值。

Math对象

```
var pi_value=Math.PI;  
var sqrt_value=Math.sqrt(15);
```

Math 对象属性

属性	描述
E	返回算术常量 e ，即自然对数的底数（约等于 2.718 ）。
LN2	返回 2 的自然对数（约等于 0.693 ）。
LN10	返回 10 的自然对数（约等于 2.302 ）。
LOG2E	返回以 2 为底的 e 的对数（约等于 1.414 ）。
LOG10E	返回以 10 为底的 e 的对数（约等于 0.434 ）。
PI	返回圆周率（约等于 3.14159 ）。
SQRT1_2	返回返回 2 的平方根的倒数（约等于 0.707 ）。
SQRT2	返回 2 的平方根（约等于 1.414 ）。

Math 对象方法

方法	描述
abs(x)	返回数的绝对值。
acos(x)	返回数的反余弦值。
asin(x)	返回数的反正弦值。
atan(x)	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)	返回从 x 轴到点 (x,y) 的角度（介于 $-\pi/2$ 与 $\pi/2$ 弧度之间）。
ceil(x)	对数进行上舍入。
cos(x)	返回数的余弦。
exp(x)	返回 e 的指数。
floor(x)	对数进行下舍入。
log(x)	返回数的自然对数（底为 e ）。
max(x,y)	返回 x 和 y 中的最高值。
min(x,y)	返回 x 和 y 中的最低值。
pow(x,y)	返回 x 的 y 次幂。
random()	返回 0 ~ 1 之间的随机数。
round(x)	把数四舍五入为最接近的整数。
sin(x)	返回数的正弦。
sqrt(x)	返回数的平方根。
tan(x)	返回角的正切。
toSource()	返回该对象的源代码。
valueOf()	返回 Math 对象的原始值。

RegExp对象

直接量: /pattern/attributes

attributes

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

方括号

方括号用于查找某个范围内的字符:

表达式	描述
[abc]	查找方括号之间的任何字符。
[^abc]	查找任何不在方括号之间的字符。
[0-9]	查找任何从 0 至 9 的数字。
[a-z]	查找任何从小写 a 到小写 z 的字符。
[A-Z]	查找任何从大写 A 到大写 Z 的字符。
[A-z]	查找任何从大写 A 到小写 z 的字符。
[adgk]	查找给定集合内的任何字符。
[^adgk]	查找给定集合外的任何字符。
(red blue green)	查找任何指定的选项。

元字符

元字符（Metacharacter）是拥有特殊含义的字符:

元字符	描述
.	查找单个字符，除了换行和行结束符。
\w	查找单词字符。
\W	查找非单词字符。
\d	查找数字。
\D	查找非数字字符。
\s	查找空白字符。
\S	查找非空白字符。
\b	匹配单词边界。
\B	匹配非单词边界。
\0	查找 NUL 字符。
\n	查找换行符。
\f	查找换页符。
\r	查找回车符。
\t	查找制表符。
\v	查找垂直制表符。
\xxx	查找以八进制数 xxx 规定的字符。
\xdd	查找以十六进制数 dd 规定的字符。
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符。

量词

量词	描述
n+	匹配任何包含至少一个 n 的字符串。
n*	匹配任何包含零个或多个 n 的字符串。
n?	匹配任何包含零个或一个 n 的字符串。
n{X}	匹配包含 X 个 n 的序列的字符串。
n{X,Y}	匹配包含 X 或 Y 个 n 的序列的字符串。
n{X,}	匹配包含至少 X 个 n 的序列的字符串。
n\$	匹配任何结尾为 n 的字符串。
^n	匹配任何开头为 n 的字符串。
?=n	匹配任何其后紧接指定字符串 n 的字符串。
?!n	匹配任何其后没有紧接指定字符串 n 的字符串。

RegExp 对象属性

属性	描述
global	RegExp 对象是否具有标志 g 。
ignoreCase	RegExp 对象是否具有标志 i 。
lastIndex	一个整数，标示开始下一次匹配的字符位置。
multiline	RegExp 对象是否具有标志 m 。
source	正则表达式的源文本。

RegExp 对象方法

方法	描述
compile	编译正则表达式。
exec	检索字符串中指定的值。返回找到的值，并确定其位置。
test	检索字符串中指定的值。返回 true 或 false 。

支持正则表达式的 String 对象的方法

方法	描述
search	检索与正则表达式相匹配的值。
match	找到一个或多个正则表达式的匹配。
replace	替换与正则表达式匹配的子串。
split	把字符串分割为字符串数组。

Number 对象

Number 对象是原始数值的包装对象。

Number 对象属性

属性

[constructor](#)

[MAX_VALUE](#)

[MIN_VALUE](#)

[NaN](#)

[NEGATIVE_INFINITY](#)

[POSITIVE_INFINITY](#)

[prototype](#)

描述

返回对创建此对象的 **Number** 函数的引用。

可表示的最大的数。

可表示的最小的数。

非数字值。

负无穷大，溢出时返回该值。

正无穷大，溢出时返回该值。

使您有能力向对象添加属性和方法。

Number 对象方法

方法

[toString](#)

[toLocaleString](#)

[toFixed](#)

[toExponential](#)

[toPrecision](#)

[valueOf](#)

描述

把数字转换为字符串，使用指定的基数。

把数字转换为字符串，使用本地数字格式顺序。

把数字转换为字符串，结果的小数点后有指定位数的数字。

把对象的值转换为指数计数法。

把数字格式化为指定的长度。

返回一个 **Number** 对象的基本数字值。

String 对象

String 对象用于处理文本（字符串）。`var s1=new String("abc")`

String 对象属性

属性	描述
constructor	对创建该对象的函数的引用
length	字符串的长度
prototype	允许您向对象添加属性和方法

String 对象方法

方法	描述
anchor()	创建 HTML 锚。
big()	用大号字体显示字符串。
blink()	显示闪动字符串。
bold()	使用粗体显示字符串。
charAt()	返回在指定位置的字符。
charCodeAt()	返回在指定的位置的字符的 Unicode 编码。
concat()	连接字符串。
fixed()	以打字机文本显示字符串。
fontcolor()	使用指定的颜色来显示字符串。
fontsize()	使用指定的尺寸来显示字符串。
fromCharCode()	从字符编码创建一个字符串。
indexOf()	检索字符串。

方法	描述
italics()	使用斜体显示字符串。
lastIndexOf()	从后向前搜索字符串。
link()	将字符串显示为链接。
localeCompare()	用本地特定的顺序来比较两个字符串。
match()	找到一个或多个正则表达式的匹配。
replace()	替换与正则表达式匹配的子串。
search()	检索与正则表达式相匹配的值。
slice()	提取字符串的片断，并在新的字符串中返回被提取的部分。
small()	使用小字号来显示字符串。
split()	把字符串分割为字符串数组。
strike()	使用删除线来显示字符串。
sub()	把字符串显示为下标。
substr()	从起始索引号提取字符串中指定数目的字符。
substring()	提取字符串中两个指定的索引号之间的字符。
sup()	把字符串显示为上标。
toLocaleLowerCase()	把字符串转换为小写。
toLocaleUpperCase()	把字符串转换为大写。
toLowerCase()	把字符串转换为小写。
toUpperCase()	把字符串转换为大写。
toSource()	代表对象的源代码。
toString()	返回字符串。
valueOf()	返回某个字符串对象的原始值。

附录4、富文本编辑框命令

下表列出了那些被支持最多的命令（第二个参数表示是否显示对话框）：

命令值	(第三个参数)	说明
backcolor	颜色字符串	设置文档的背景颜色
bold	null	将选择的文本转换为粗体
copy	null	将选择的文本复制到剪贴板
createlink	URL字符串	将选择的文本转换成一个链接，指向指定的URL
cut	null	将选择的文本剪切到剪贴板
delete	null	删除选择的文本
fontname	字体名称	将选择的文本修改为指定字体
fontsize	1~7	将选择的文本修改为指定字体大小
forecolor	颜色字符串	将选择的文本修改为指定的颜色
formatblock	要包围当前文本块的HTML标签；如<h1>	使用指定的HTML标签来格式化选择的文本块
indent	null	缩进文本
inserthorizontalrule	null	在插入字符处插入一个<hr>元素
insertimage	图像的URL	在插入字符处插入一个图像
insertorderedlist	null	在插入字符处插入一个元素
insertunorderedlist	null	在插入字符处插入一个元素
insertparagraph	null	在插入字符处插入一个<p>元素
italic	null	将选择的文本转换成斜体
justifycenter	null	将插入光标所在文本块居中对齐
justifyleft	null	将插入光标所在文本块左对齐
outdent	null	凸排文本（减少缩进）
paste	null	将剪贴板中的文本粘贴到选择的文本
removeformat	null	移除文本块的块级格式。这是撤销formatblock命令的操作
selectall	null	选择文档中的所有文本
underline	null	为选择的文本添加下划线
unlink	null	移除文本的链接。这是撤销createlink命令的操作

[参考1](#) [参考2](#) [参考3](#) [参考4](#)

附录5、元素的其它操作

<code>focus()</code>	让输入类元素得焦点 htm5。
<code>scrollIntoView()</code>	滚动页面让元素可见；
<code>contains()</code>	是否为后代元素；
<code>insertAdjacentHTML()</code>	在元素的开始标签和结束标签的前后插入元素。
<code>compareDocumentPosition()</code>	比较位置：1-无关 2-居前 3-居后 8-包含 16-被包含。

附录6、DOM事件和方法

• 鼠标事件

属性	描述	DOM
onclick	当用户点击某个对象时调用的事件句柄。	2
oncontextmenu	在用户点击鼠标右键打开上下文菜单时触发	
ondblclick	当用户双击某个对象时调用的事件句柄。	2
onmousedown	鼠标按钮被按下。	2
onmouseenter	当鼠标指针移动到元素上时触发。	2
onmouseleave	当鼠标指针移出元素时触发	2
onmousemove	鼠标被移动。	2
onmouseover	鼠标移到某元素之上。	2
onmouseout	鼠标从某元素移开。	2
onmouseup	鼠标按键被松开。	2

• 键盘事件

属性	描述	DOM
onkeydown	某个键盘按键被按下。	2
onkeypress	某个键盘按键被按下并松开。	2
onkeyup	某个键盘按键被松开。	2

DOM: 指明使用的 DOM 属性级别。

• 框架/对象（Frame/Object）事件

属性	描述	DOM
onabort	图像的加载被中断。(<object>)	2
onbeforeunload	该事件在即将离开页面（刷新或关闭）时触发	2
onerror	在加载文档或图像时发生错误。(<object>, <body>和 <frameset>)	
onhashchange	该事件在当前 URL 的锚部分发生修改时触发。	
onload	一张页面或一幅图像完成加载。	2
onpageshow	该事件在用户访问页面时触发	
onpagehide	该事件在用户离开当前网页跳转到另外一个页面时触发	
onresize	窗口或框架被重新调整大小。	2
onscroll	当文档被滚动时发生的事件。	2
onunload	用户退出页面。(<body> 和 <frameset>)	2

- 表单事件

属性	描述	DOM
onblur	元素失去焦点时触发	2
onchange	该事件在表单元素的内容改变时触发(<input>, <keygen>, <select>, 和 <textarea>)	2
onfocus	元素获取焦点时触发	2
onfocusin	元素即将获取焦点时触发	2
onfocusout	元素即将失去焦点时触发	2
oninput	元素获取用户输入时触发	3
onreset	表单重置时触发	2
onsearch	用户向搜索域输入文本时触发 (<input="search">)	
onselect	用户选取文本时触发 (<input> 和 <textarea>)	2
onsubmit	表单提交时触发	2

- 剪贴板事件

属性	描述	DOM
oncopy	该事件在用户拷贝元素内容时触发	
oncut	该事件在用户剪切元素内容时触发	
onpaste	该事件在用户粘贴元素内容时触发	

- 拖动事件

事件	描述	DOM
ondrag	该事件在元素正在拖动时触发	
ondragend	该事件在用户完成元素的拖动时触发	
ondragenter	该事件在拖动的元素进入放置目标时触发	
ondragleave	该事件在拖动元素离开放置目标时触发	
ondragover	该事件在拖动元素在放置目标上时触发	
ondragstart	该事件在用户开始拖动元素时触发	
ondrop	该事件在拖动元素放置在目标区域时触发	

- 打印事件

属性	描述	DOM
onafterprint	该事件在页面已经开始打印，或者打印窗口已经关闭时触发	
onbeforeprint	该事件在页面即将开始打印时触发	

• 多媒体（Media）事件

事件	描述
onabort	事件在视频/音频（audio/video）终止加载时触发。
oncanplay	事件在用户可以开始播放视频/音频（audio/video）时触发。
oncanplaythrough	事件在视频/音频（audio/video）可以正常播放且无需停顿和缓冲时触发。
ondurationchange	事件在视频/音频（audio/video）的时长发生变化时触发。
onemptied	当期播放列表为空时触发
onended	事件在视频/音频（audio/video）播放结束时触发。
onerror	事件在视频/音频（audio/video）数据加载期间发生错误时触发。
onloadeddata	事件在浏览器加载视频/音频（audio/video）当前帧时触发触发。
onloadedmetadata	事件在指定视频/音频（audio/video）的元数据加载后触发。
onloadstart	事件在浏览器开始寻找指定视频/音频（audio/video）触发。
onpause	事件在视频/音频（audio/video）暂停时触发。
onplay	事件在视频/音频（audio/video）开始播放时触发。
onplaying	事件在视频/音频（audio/video）暂停或者在缓冲后准备重新开始播放时触发。
onprogress	事件在浏览器下载指定的视频/音频（audio/video）时触发。
onratechange	事件在视频/音频（audio/video）的播放速度发送改变时触发。
onseeked	事件在用户重新定位视频/音频（audio/video）的播放位置后触发。
onseeking	事件在用户开始重新定位视频/音频（audio/video）时触发。
onstalled	事件在浏览器获取媒体数据，但媒体数据不可用时触发。
onsuspend	事件在浏览器读取媒体数据中止时触发。
ontimeupdate	事件在当前的播放位置发送改变时触发。
onvolumechange	事件在音量发生改变时触发。
onwaiting	事件在视频由于要播放下一帧而需要缓冲时触发。

• 动画事件

事件	描述	DOM
animationend	该事件在 CSS 动画结束播放时触发	
animationiteration	该事件在 CSS 动画重复播放时触发	
animationstart	该事件在 CSS 动画开始播放时触发	

• 过渡事件

事件	描述	DOM
transitionend	该事件在 CSS 完成过渡后触发。	

• 其他事件

事件	描述	DOM
onmessage	该事件通过或者从对象(WebSocket, Web Worker, Event Source 或者子 frame 或父窗口)接收到消息时触发	
onmousewheel	已废弃。使用 onwheel 事件替代	
ononline	该事件在浏览器开始在线工作时触发。	
onoffline	该事件在浏览器开始离线工作时触发。	
onpopstate	该事件在窗口的浏览历史（history 对象）发生改变时触发。	
onshow	该事件当 <menu> 元素在上下文菜单显示时触发	
onstorage	该事件在 Web Storage(HTML 5 Web 存储)更新时触发	
ontoggle	该事件在用户打开或关闭 <details> 元素时触发	
onwheel	该事件在鼠标滚轮在元素上下滚动时触发	

• 常用事件

<code>onclick</code>	元素被点击
<code>onabort</code>	图像加载被中断
<code>onblur</code>	元素失去焦点
<code>onchange</code>	用户改变域的内容
<code>onerror</code>	当加载文档或图像时发生某个错误
<code>onfocus</code>	元素获得焦点
<code>onload</code>	某个页面或图像被完成加载
<code>onreset</code>	重置按钮被点击
<code>onresize</code>	窗口或框架被调整尺寸
<code>onselect</code>	文本被选定
<code>onsubmit</code>	提交按钮被点击
<code>onunload</code>	用户退出页面
<code>onpropertychange</code>	元素属性改变 (<code>event.propertyName</code> 设置或返回元素的变化了的属性的名称)

•鼠标事件

▫ 事件列举：

```
<div onclick="alert('click')" > hello </div>
```

click	单击	dblclick	双击	mousedown	按下	mouseup	放开
mouseenter	进入	mouseout	离开	mousemove	移动	mouseover	悬浮

▫ 事件属性：

event.screenX, event.screenY	鼠标在窗口的坐标
event.clientX,event.clientY	鼠标在客户区的坐标
event.offsetX, event.offsetY	鼠标在点击对象上的坐标
event.pageX,event.pageY	鼠标在页面中的坐标
event.x,event.y	相对于上级元素的坐标(默认为BODY)
event.button	按下的鼠标键: 0 没按键 1 按左键 2 按右键 3 按左右键 4 按中间键 5 按左键和中间键 6 按右键和中间键 7 按所有的键
event.fromElement	鼠标移动离开的对象
event.toElement	鼠标移动进入的对象
event.srcElement	触发事件(鼠标点击)的元素
event.target	冒泡事件中的最小范围元素
event.currentTarget	当前事件的元素（同this）

• 键盘事件

keydown
keyup, keypress
event.keyCode
event.charCode
event.altKey, event.ctrlKey, event.shiftKey:

按下任意键
按下字符键，长按则连续触发
按键内码（扫描码）
按键字符(可显示)
alt, ctrl或shift的键是否按下，按下为true

• 触摸屏事件

touchstart
touchend
touchmove
touchenter
touchcancel
touchleave
touches
targetTouches
changeTouches
orientationchange
devicemotion

开始触摸
结束触摸。触摸位置的参数与鼠标事件一样。
手指移动。
手指进入目标的有效区域
中断触摸或手指离开有效区域
中断还在触摸但是离开了有效区域
跟踪Touch对象的数组
特定于时间目标的Touch对象数组
自从上次触摸以来发生了什么改变的Touch对象数组
方向：0-肖像模式 90-左旋90度 90-右旋90度
设备正在移动

• 手势事件

gesturestart
gesturechange
gestureend
event.scale
event.orientation

两个手指均触摸时发生
至少有一个手指位置发生变化。
至少有一个手指从触摸屏移开。
两指间距离,从1开始，值越大距离越大
手指变化的旋转角度，正(负)值为正(逆)时针旋转

```
<head>
<script type="text/javascript">
    function bind(fn, context) {
        return function () {
            return fn.apply(context, arguments);
        };
    }
</script>
</head>
<body>
<p id="p1">请点击该文本</p>
<p id="p2">this is another paragraph</p>
<script type="text/javascript">
var handler = {
    message: "Event handled",
    handleClick: function (event) {
        alert(this.message);
    }
};
var btn = document.getElementById("p1");
btn.onclick = bind(handler.handleClick, handler);
</script>
</body>
```

附录7、事件对象(event)

• 常量

静态变量	描述	DOM
CAPTURING-PHASE	当前事件阶段为捕获阶段(3)	1
AT-TARGET	当前事件是目标阶段,在评估目标事件(1)	2
BUBBLING-PHASE	当前的事件为冒泡阶段 (2)	3

• 属性

属性	描述	DOM
bubbles	返回布尔值，指示事件是否是起泡事件类型。	2
cancelable	返回布尔值，指示事件是否可拥可取消的默认动作。	2
currentTarget	返回其事件监听器触发该事件的元素。	2
eventPhase	返回事件传播的当前阶段。	2
target	返回触发此事件的元素（事件的目标节点）。	2
timeStamp	返回事件生成的日期和时间。	2
type	返回当前 Event 对象表示的事件的名称。	2

• 方法

方法	描述	DOM
initEvent()	初始化新创建的 Event 对象的属性。	2
preventDefault()	通知浏览器不要执行与事件关联的默认动作。	2
stopPropagation()	不再派发事件。	2

附录8、 鼠标/键盘事件对象

属性	描述	DOM
altKey	返回当事件被触发时, "ALT" 是否被按下。	2
button	返回当事件被触发时, 哪个鼠标按钮被点击。	2
clientX	返回当事件被触发时, 鼠标指针的水平坐标。	2
clientY	返回当事件被触发时, 鼠标指针的垂直坐标。	2
ctrlKey	返回当事件被触发时, "CTRL" 键是否被按下。	2
Location	返回按键在设备上的位置	3
charCode	返回onkeypress事件触发键值的字母代码。	2
key	在按下按键时返回按键的标识符。	3
keyCode	返回onkeypress事件触发的键的值的字符代码, 或者 onkeydown 或 onkeyup 事件的键的代码。	2
which	返回onkeypress事件触发的键的值的字符代码, 或者 onkeydown 或 onkeyup 事件的键的代码。	2
metaKey	返回当事件被触发时, "meta" 键是否被按下。	2
relatedTarget	返回与事件的目标节点相关的节点。	2
screenX	返回当某个事件被触发时, 鼠标指针的水平坐标。	2
screenY	返回当某个事件被触发时, 鼠标指针的垂直坐标。	2
shiftKey	返回当事件被触发时, "SHIFT" 键是否被按下。	2

方法	描述	W3C
<code>initMouseEvent()</code>	初始化鼠标事件对象的值	2
<code>initKeyboardEvent()</code>	初始化键盘事件对象的值	3

附录9、其它对象的方法

- 目标事件对象的方法

方法	描述	DOM
addEventListener()	允许在目标事件中注册监听事件(IE8 = attachEvent())	2
dispatchEvent()	允许发送事件到监听器上 (IE8 = fireEvent())	2
removeEventListener()	运行一次注册在事件目标上的监听事件(IE8 = detachEvent())	2

- 事件监听对象的方法

方法	描述	DOM
handleEvent()	把任意对象注册为事件处理程序	2

- 文档事件对象的方法

方法	描述	DOM
createEvent()		2

附录10、IE8.0及更早版本的事件

```
var p1=document.getElementById("p1");
var f1=function(){alert("hello");};
if(p1.addEventListener){
    p1.addEventListener("click", function(event){
        alert(event.type+":"+event.clientX+","+event.clientY+"
            +event.screenX+","+event.screenY);
    });
}
else{
    p1.attachEvent("onclick", function(){ var event = window.event;
        alert(event.type+":"+event.clientX+","+event.clientY
            +event.screenX+","+event.screenY);
    });
}
```

`p1.detachEvent("onclick", function(){ ... });` //删除事件处理程序

`window.event.returnValue = false` // IE9之前的版本用于取消默认行为

`window.event.cancelBubble = true;` // IE9之前的版本用于阻止事件传播：

可以做一个跨浏览器平台的事件绑定函数：

```
var addMyEvent = function(obj, ev, fn){  
    if(obj.addEventListener){  
        obj.addEventListener(ev, fn, false);  
    }else if(obj.attachEvent){  
        obj.attachEvent("on"+ev, fn);  
    }else{  
        obj["on"+ev] = fn;  
    }  
}
```


附录11、自定义属性的特征

自定义属性默认是可枚举的，可以通过defineProperty重新定义进行改变。

```
var person1={name:"Nicholas"};
alert(person1.propertyIsEnumerable("name"));           //true
Object.defineProperty(person1,"name",{enumerable:false});
alert(person1.propertyIsEnumerable("name"));           //false
Object.defineProperty(person1,"name",{enumerable:true});
alert(person1.propertyIsEnumerable("name"));           //true
```

如果定义为不可配置的，则不能改变或删除对象的属性。通过定义为不可写入的，则不能改变属性值。设置禁止扩展后将不允许增加新属性。如果要求只能读写属性值，不可增删属性和改变属性特征，则要设置为被封印。如果不允许属性进行任何更改，其只能读出属性值，则可冻结对象。冻结对象也是不可扩展和被封印的。

```
Object.defineProperty(person1,"name",{configurable:false});
Object.defineProperty(person1,"name",{writable:false});
Object.preventExtensions(person1);
alert(Object.isExtensible(person1)); //false
Object.seal(person1);
alert(Object.isSealed(person1));     //true
Object.freeze(person1);
alert(Object.isFrozen(person1));     //true
```

通过描述子得到属性特征:

```
var person1={name:"Nicholas"};
var desc = Object.getOwnPropertyDescriptor(person1,"name");
alert(desc.enumerable);
alert(desc.configurable);
alert(desc.writable);
alert(desc.value);
```

采用defineProperty定义一个属性:

```
var person1 = {};
Object.defineProperty(person1,"name", {
    value:"Nicholas",
    enumerable:true,
    configurable:true,
    writable:true
})
alert(person1.name);    // Nicholas
```

采用defineProperties可以一次定义多个属性，还可以定义访问器属性(get和set函数)：

```
var person2 = {};  
Object.defineProperties(person2,{  
  _name:{  
    value:"Nicholas",  
    enumerable:true,  
    configurable:true,  
    writable:true  
  },  
  name:{  
    get:function(){  
      return this._name;  
    },  
    set:function(val){  
      this._name = val;  
    },  
    enumerable: true,  
    configurable: true  
  }  
})  
person2.name="David";  
alert(person2.name);
```

数据属性和访问器属性

附录12、通过原型继承

JavaScript的对象的原型属性默认都是指向Object.prototype对象。Object.prototype对象的原型属性为空。当查找对象属性时，先到本对象的自有属性中查找，然后沿着原型链进行查找。

```
var book={  
    title:"My Road",  
};
```

// 上面定义与下面定义相同，Object.create的第一个参数赋值给对象的原型属性
// 第二个参数用于定义其它属性

```
var book=Object.create(Object.prototype,{  
    title:{  
        configurable:true,  
        enumerable:true,  
        value:"My Road...",  
        writable:true  
    }  
})  
alert(book.title);
```

```

// var person1={ name:"Greg", sayName: function(){alert(this.name);}}; //与下面定义等同
var person1=Object.create(Object.prototype,{ // 改为null, 下面的toString()将出错
    name:{
        configurable:true,
        enumerable:true,
        value:"Greg",
        writable:true
    },
    sayName:{
        configurable:true,
        enumerable:true,
        value:function(){alert(this.name);},
        writable:true
    }
});
var person2=Object.create(person1,{
    //constructor:person2,
    name:{
        configurable:true,
        enumerable:true,
        value:"Nicholas",
        writable:true
    }
});
person1.sayName(); //Greg
person2.sayName(); //Nicholas
alert(person2.toString()); //[object Object].

```

如果只继承原型的属性，则可以使用原型赋值：Person.prototype=Parent.prototype;
如果Person要使用Parent的构造函数，可以对Parent使用call的方法。

```
function Person(name){
    this.name = name;
}
Person.prototype.sayName=function(){alert(this.name);};
function Person1(name){
    Person.call(this,name);
}
Person1.prototype = Person.prototype;           // 可以用Object.create创建
var person1 = new Person1("Greg");
person1.sayName();                               // Greg
alert(person1.hasOwnProperty("Name"));           // true
```

利用call可以直接调用双亲中的方法：

```
function Person1(name){
    Person.call(this,name);
    Person.prototype.sayName.call(this,name);
}
```

附录13、私有成员

利用立调函数形成私有成员的方法（利用了闭包）：

```
var person =  
  (function(){  
    var age = 25; // 私有成员，一般采用下划线命名_age  
    return {  
      name: "Nicholas",  
      getAge: function(){  
        return age;  
      },  
      growOlder: function(){  
        age++;  
      }  
    }  
  })()  
;  
console.log(person.name); // Nicholas  
console.log(person.getAge()); // 25  
person.age = 125; // 不能访问age  
console.log(person.getAge()); // 25  
person.growOlder();  
console.log(person.getAge()); // 26
```

也可以先定义好所有的内部数据和函数，然后再定义要返回对象（暴露模块模式）。

```
var person =  
    (function(){  
        var age = 25;  
        function getAge(){  
            return age;  
        };  
        function growOlder(){  
            age++;  
        };  
        return {  
            name:"Nicholas",  
            getAge:getAge,  
            growOlder:growOlder  
        };  
    })();  
console.log(person.name);           // Nicholas  
console.log(person.getAge());       // 25  
person.age = 125;  
console.log(person.getAge());       // 25  
person.growOlder();  
console.log(person.getAge());       // 26
```


利用构造函数形成私有成员:

```
function Person(name){
    var age = 25;
    this.name = name;
    this.getAge=function(){
        return age;
    }
    this.growOlder=function(){
        age++;
    }
}

var person1 = new Person("Nicholas");
var person2 = new Person("Greg");

console.log(person1.name);           // Nicholas
console.log(person2.name);           // Greg
console.log(person1.getAge());        // 25
person1.age = 125;
console.log(person1.getAge());        // 25
person1.growOlder();
console.log(person1.getAge());        // 26
console.log(person2.getAge());        // 25
```

利用原型共享函数，也共享了私有变量：

```
var Person = (function(){
    var age = 25;
    function InnerPerson(name){
        this.name = name;
    };
    InnerPerson.prototype.getAge=function(){
        return age;
    };
    InnerPerson.prototype.growOlder=function(){
        age++;
    };
    return InnerPerson;
})();
var person1 = new Person("Nicholas");
var person2 = new Person("Greg");
console.log(person1.name);           // Nicholas
console.log(person2.name);           // Greg
person1.age = 125;
console.log(person1.getAge());        // 25
person1.growOlder();
console.log(person1.getAge());        // 26
console.log(person2.getAge());        // 26
```

附录14、keycode对照表

字母和数字键的键码值(keyCode)							
按键	键码	按键	键码	按键	键码	按键	键码
A	65	J	74	S	83	1	49
B	66	K	75	T	84	2	50
C	67	L	76	U	85	3	51
D	68	M	77	V	86	4	52
E	69	N	78	W	87	5	53
F	70	O	79	X	88	6	54
G	71	P	80	Y	89	7	55
H	72	Q	81	Z	90	8	56
I	73	R	82	0	48	9	57

数字键盘上的键的键码值(keyCode)				功能键键码值(keyCode)			
按键	键码	按键	键码	按键	键码	按键	键码
0	96	8	104	F1	112	F7	118
1	97	9	105	F2	113	F8	119
2	98	*	106	F3	114	F9	120
3	99	+	107	F4	115	F10	121
4	100	Enter	108	F5	116	F11	122
5	101	-	109	F6	117	F12	123
6	102	.	110				
7	103	/	111				

控制键键码值(keyCode)							
按键	键码	按键	键码	按键	键码	按键	键码
BackSpace	8	Esc	27	Right Arrow	39	_	189
Tab	9	Spacebar	32	Dw Arrow	40	.>	190
Clear	12	Page Up	33	Insert	45	/?	191
Enter	13	Page Down	34	Delete	46	~`	192
Shift	16	End	35	Num Lock	144	[{	219
Control	17	Home	36	;	186	\	220
Alt	18	Left Arrow	37	=+	187]}	221
Cape Lock	20	Up Arrow	38	,<	188	"	222

多媒体键码值(keyCode)							
按键	键码	按键	键码	按键	键码	按键	键码
音量加	175						
音量减	174						
停止	179						
静音	173						
浏览器	172						
邮件	180						
搜索	170						
收藏	171						

附录15、height、width与scroll

height
width

元素内容的高和宽

clientHeight
clientWidth

元素的显示窗口的高和宽（不包含滚动条）

offsetHeight
offsetWidth

元素除了margin之外的高和宽（包含滚动条）

offsetLeft
offsetTop

元素的边框（包含滚动条）与父元素之间的距离

scrollHeight
scrollWidth

元素内容的总高度和总宽度(包含滚动在窗口外的部分)。

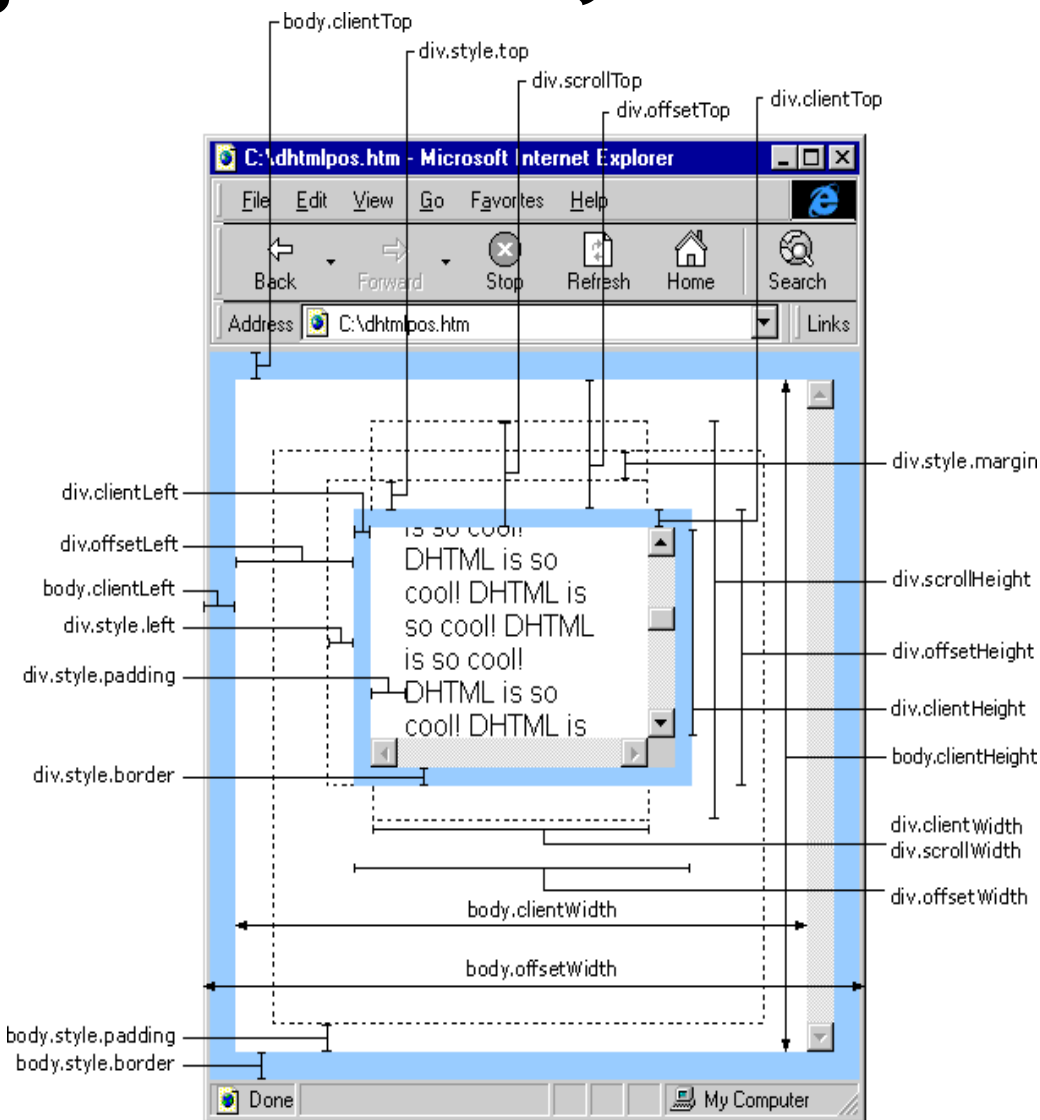
scrollLeft
scrollTop

因为滚动隐藏在内容区域左侧（或顶部）的像素数。设置它们会造成滚动而隐藏一部分内容

scrollIntoView()

所有html元素都有这个方法，如果参数为true（默认），则滚动元素使元素的顶部与窗口对齐，如果参数为false，则会尽量多地显示元素，超出时底部会对齐

window.scroll(x,y)或**window.scrollTo(x,y)**让文档元素左滚x像素，上滚y像素，默认左滚和上滚都是0。



附录16、获得元素的样式属性的值

//连接表示法变成驼峰表示法，例如：data_src变为dataSrc

```
function camelize(s) {  
    return s.replace(/-(\w)/g, function(strMatch, p1) {  
        return p1.toUpperCase();  
    });  
};
```

//取得元素element样式属性property的值

```
function getStyle(element, property) {  
    if (arguments.length != 2) return false;  
    var value = element.style[camelize(property)];  
    if (!value) {  
        if (document.defaultView && document.defaultView.getComputedStyle) {  
            var css = document.defaultView.getComputedStyle(element, null);  
            value = css ? css.getPropertyValue(property) : null;  
        } else if (element.currentStyle) {  
            value = element.currentStyle[camelize(property)];  
        }  
    }  
    return value == 'auto' ? '' : value;  
};
```

访问z-index和background-color:

```
p.style.zIndex='998';  
p.style.backgroundColor="gray";
```

附录17、表单数据串行化

如果采用AJAX技术实现post功能，就要对输入数据进行串行化。串行化就是把form中每个输入字段的值全部取到并形成如下key-value格式的数据：

`x1=v1&x2=v2&x3=v3...`

如果x和v使用了汉字，则要使用函数encodeURIComponent()进行编码。利用ajax技术可以把这个串行化的数据提交给后台处理。

下面是一个对表单(form)的串行化函数：

```
function serialize(form){
    var parts = [], field = null,i,len,j,optLen,option,optValue;
    for (i=0, len=form.elements.length; i < len; i++){
        field = form.elements[i];
        switch(field.type){
            ... // 见下页
        }//switch
    }//for
    return parts.join("&");
}//function
```

```
switch(field.type){
  case "select-one":
  case "select-multiple":
    if (field.name.length){
      for (j=0, optLen = field.options.length; j < optLen; j++){
        option = field.options[j];
        if (option.selected){
          optValue = "";
          if (option.hasAttribute){
            optValue = (option.hasAttribute("value")?
              option.value: option.text);
          }else {
            optValue = (option.attributes["value"].specified?
              option.value:option.text);
          } //if
          parts.push(encodeURIComponent(field.name) + "=" +
            encodeURIComponent(optValue));
        } //if
      } //for
    } //if
    break;
```

```
case undefined:           //字段集
case "file":              //文件输入
case "submit":            //提交按钮
case "reset":             //重置按钮
case "button":            //自定义按钮
    break;
case "radio":              //单选按钮
case "checkbox":             //复选框
    if (!field.checked){ break;}
/* 执行默认操作 */
default:                  //不包含没有名字的表单字段
    if (field.name.length){
        parts.push(encodeURIComponent(field.name)
                    + "=" + encodeURIComponent(field.value));
    }
}
} //switch
```


附录18、jQuery的ready函数定义

```
document.ready = function (callback) { //callback为要执行的函数
    if (document.addEventListener) { //兼容FF,Google
        document.addEventListener('DOMContentLoaded', function () {
            document.removeEventListener('DOMContentLoaded',
                arguments.callee, false);
            callback();
        }, false)
    }
    else if (document.attachEvent) { //兼容IE
        document.attachEvent('onreadystatechange', function () {
            if (document.readyState == "complete") {
                document.detachEvent("onreadystatechange",
                    arguments.callee);
                callback();
            }
        })
    }
    else if (document.lastChild == document.body) {
        callback();
    }
}
```

附录19、console对象

- `console.log(object[, object, ...])`
在浏览器控制台显示`object`的值，可以进行格式化输出。

```
var x='abcde', y=100;  
console.log("x=%s, y=%d", x, y);  
console.log("x=", x, ", y=", y);
```
- 具有类似功能的控制台函数还有`console.info()`、`console.warn()`、`console.error()`和`console.debug()`，它们之间的区别只是图标和分类不同。`error()`会抛出一个 `js` 异常，与 `throw new Error()` 产生的效果相同。`debug()`包括一个指向该行代码位置的超链接。
- `console.assert(expression[, object, ...])`
测试一条表达式是否为真，不为真时将抛出异常（断言失败）。
- `console.dir(object)`
输出一个对象的全部属性（输出结果类似于 DOM 面板中的样式）。
- `console.dirxml(node)`
输出一个 HTML 或者 XML 元素的结构树，点击结构树上的节点进入到 HTML 面板。
- `console.trace()`
输出 Javascript 执行时的堆栈追踪。

- `console.group(object[, object, ...])`
输出消息并打开一个缩进输出的嵌套块，`console.groupEnd()` 结束这个块。`groupCollapsed()`产生收起的嵌套块。
- `console.time(name)`
命名计时器，用 `console.timeEnd(name)`结束计时并输出它们之间执行的代码所花费的时间（毫秒）。
- `console.profile([title])`
与 `profileEnd()`配合使用，用来做性能测试，与 `console` 面板上 `profile` 按钮的功能完全相同。
- `console.count([title])`
输出本语句执行的次数，`title`作为显示标题。
- `console.clear()`
清空控制台

```
console.time('test');    // test为定时器名字
for (var i = 0; i < 1000; i++) { console.log(i); };
console.timeEnd('test')
```

test: 42.583984375ms

附录20、控制台的调试命令

在浏览器控制台的输出面板中除了可以运行javascript 代码，还可以利用一些内置命令帮助JS的调试。

- `$(id)` 返回一个给定 id 的元素。
- `$$(selector)` 返回给定的 css 选择器匹配到的一组元素。
- `$x(xpath)` 返回给定的 XPath 表达式匹配到的一组元素。
- `$0` 在 HTML 面板中选中的元素。
- `$1` 上一次在 HTML 面板中选中的元素。
- `$n(index)` 访问最近被选中过的五个元素之一，index 的范围： 0 - 4。
- `dir(object)` 同 `console.dir(object)`。
- `dirxml(node)` 同 `console.dirxml(node)`。
- `clear()` 同 `console.clear()`。
- `inspect(object[, tabName])()` 在合适的（或一个指定的） tab 中检视一个对象。
- `keys(object)` 返回一个对象的所有属性的键。
- `values(object)` 返回一个对象的所有属性的值。
- `debug(fn)` 在函数第一行添加一个断点，使用 `undebug(fn)` 移除断点。
- `monitor(fn)` 开启一个函数的调用日志，使用 `unmonitor(fn)` 关闭该功能。
- `monitorEvents(object[, types])` 设置元素的触发事件。例如，`monitorEvents($0,['click'])` 被执行后，当HTML面板中被选中元素触发事件click时控制台会显示该对象。如果不指定第二个参数，将对所有事件进行记录。
- `profile([title])` 同 `console.profile([title])`

附录21、JS编程规范

- 使用空格代替tab (2个空格替代一个tab)
- 每个语句必须以分号结尾

[谷歌](#) [Airbnb](#) [其他](#)

```
// bad
let luke = {}
let leia = {}
[luke, leia].forEach(jedi => jedi.father = 'vader')
// good
let luke = {};
let leia = {};
[luke, leia].forEach((jedi) => { jedi.father = 'vader'; });
```

- 除非规范制定完成，最好不要使用ES6模块化（export和import）

```
//----- lib.js -----
export function square(x) {
  return x * x;
}
export function diag(x, y) {
  return sqrt(square(x) + square(y));
}
//----- main.js -----
import { square, diag } from 'lib';
```

- 不推荐对代码进行水平对齐。

```
// bad
{
  tiny:    42,
  longer: 435,
};
// good
{
  tiny: 42,
  longer: 435,
};
```

- 使用**const**或**let**而不是**var**来声明局部变量。
- 优先使用箭头函数。
- 使用模板字符串取代连接字符串。

```
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}
```

```
// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

- 不要使用续行符"\\"分割长字符串。

```
// bad
const longString = 'This is a very long string that \
far exceeds the 80 column limit.';

// good
const longString = 'This is a very long string that '
  + 'far exceeds the 80 column limit.';
```

- 不要使用eval或是Function(...string)结构。
- ES6的三种for语句，最好使用for...of...。
- 常量命名应该使用全大写格式，并用下划线分割，常量函数名还是使用驼峰命名法。
- 每次只声明一个变量。

```
// bad
let a = 1, b = 2, c = 3;

// good
let a = 1;
let b = 2;
let c = 3;
```

- 只允许使用单引号包裹普通字符串，禁止使用双引号。如果字符串中包含单引号字符，应该使用模板字符串。

```
// bad
let directive = "No identification of self or mission."
// bad
let saying = 'Say it ain\u0027t so.'; // good
let directive = 'No identification of self or mission.';
// good
let saying = `Say it ain't so`;
```

- 使用计算属性名创建具有动态属性名的对象。

```
function getKey(k) {
  return `a key named ${k}`;
}
```

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;
```

```
// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```


- 使用对象方法的简写。

// bad

```
const atom = {  
  value: 1,  
  addValue: function (value) {  
    return atom.value + value;  
  },  
};
```

// good

```
const atom = {  
  value: 1,  
  addValue(value) {  
    return atom.value + value;  
  },  
};
```

- 使用对象属性值的简写。如果有很多属性将简写属性写在前面。

```
const lukeSkywalker = 'Luke Skywalker';
```

// bad

```
const obj = {  
  lukeSkywalker: lukeSkywalker,  
};
```

// good

```
const obj = {  
  lukeSkywalker,  
};
```

- 浅拷贝对象的时候使用 ... 操作符而不是 Object.assign。

```
// very bad
const original = { a: 1, b: 2 };
const copy = Object.assign(original, { c: 3 }); // this mutates `original` ☹️
delete copy.a; // so does this

// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2, c: 3 }

// good
const original = { a: 1, b: 2 };
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }

const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

- 使用对象解构而不是数组解构来实现多个返回值。

```
function prn(...args) { // 可变参数的推荐方法, 不用arguments
    for(var i=0; i<args.length; i++) {
        document.write(args[i]+" ");
    }
}
```

```
const arr=["left", "right", "top", "bottom"];
```

```
// bad
```

```
function processInput1(input) {
    const [left, right, top, bottom]=input;
    return [left, right, top, bottom];
}
```

```
const [left1, _, top1]=processInput1(arr); // 循序选取其中项目
prn(left1, top1);    // left top
```

```
// good
```

```
function processInput2(input) {
    const [left2, right2, top2, bottom2]=input;
    return {left2, right2, top2, bottom2};
}
```

```
const {left2, top2} = processInput2(arr); // 任意选取其中项目
prn(left2, top2);    // left top
```

- 变量不要链式赋值

```
(function example() {  
    // JavaScript 将其解析为  
    // let a = ( b = ( c = 1 ) );  
    // let关键字只适用于变量a;  
    // 变量b和c变成了全局变量。  
    let a = b = c = 1;  
})();
```

```
console.log(a); // 抛出 ReferenceError (引用错误)  
console.log(b); // 1  
console.log(c); // 1
```

```
// good  
(function example() {  
    let a = 1;  
    let b = a;  
    let c = a;  
})();
```

```
console.log(a); // 抛出 ReferenceError (引用错误)  
console.log(b); // 抛出 ReferenceError (引用错误)  
console.log(c); // 抛出 ReferenceError (引用错误)
```

```
// 同样适用于 `const`
```

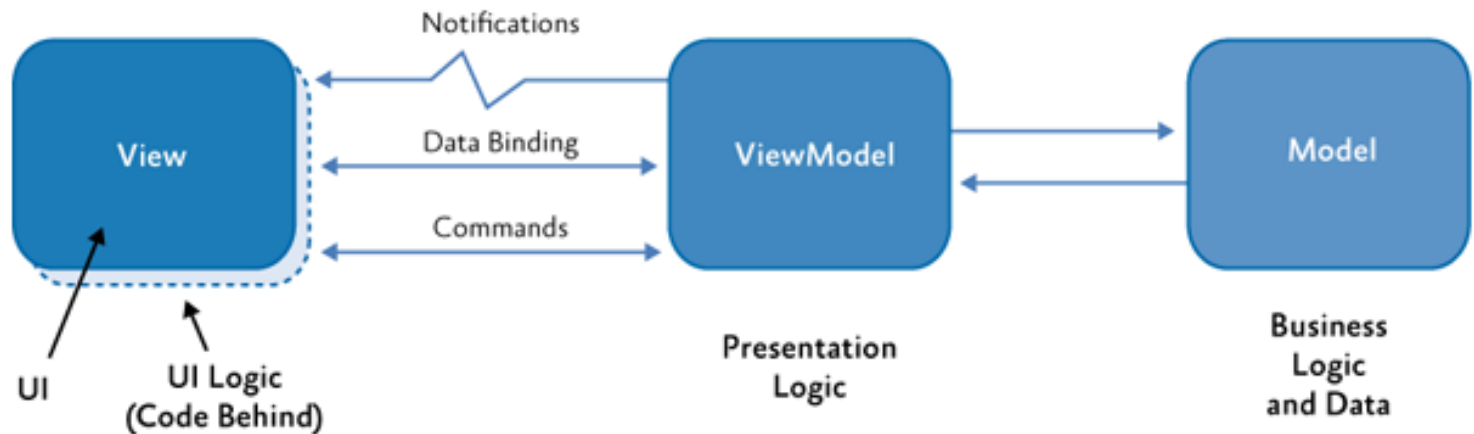
附录22、MVVM

[参考](#)

MVVM(Model、View、ViewModel)模式和MVC(Model、View、Controller)模式一样，主要目的是分离视图（View）和模型（Model），其优点：

- **低耦合**。当View变化的时候Model可以不变，反之亦然，一个ViewModel可以绑定到不同的“View”上，。
- **可重用性**。可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑。
- **独立开发**。开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面设计。
- **可测试**。界面素来是比较难于测试的，而现在测试可以针对ViewModel来写。

The MVVM classes and their interactions



Model、View、ViewModel的分工:

• View

负责界面和显示，不直接和Model交互，只能通过DataContext和ViewModel绑定。控件可以和ViewModel的公共属性绑定，Update需要双向绑定。

控件可以触发Behavior/Command调用ViewModel的方法，Command是View到ViewModel的单向通讯（View中触发事件，ViewModel中处理事件）

• ViewModel

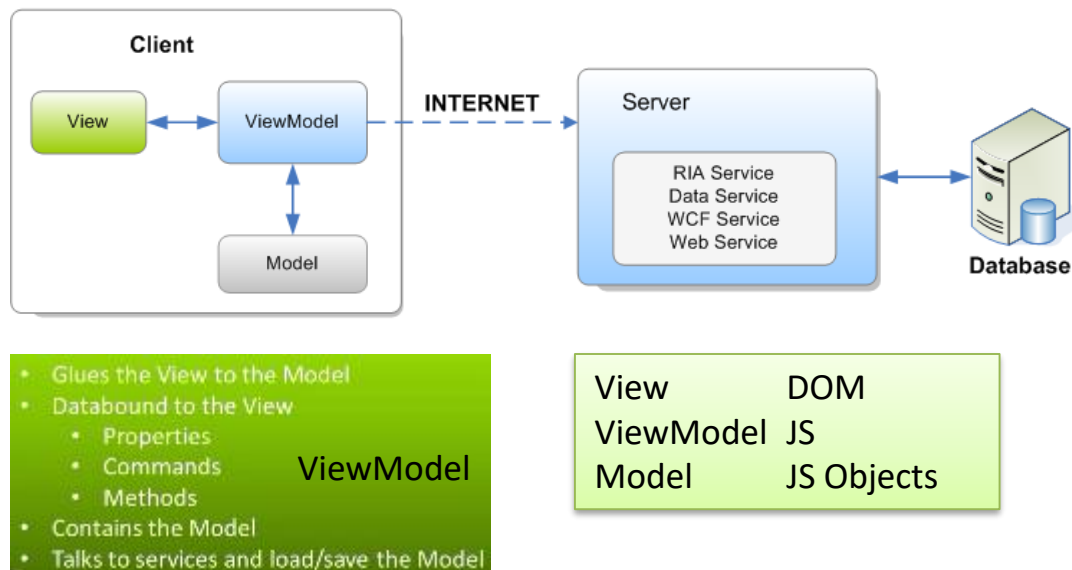
负责界面逻辑和模型数据封装，维护视图状态，对Behavior/Command事件响应，绑定的属性定义，实现属性或集合的变动通告等。

ViewModel是Model的继承类，是View和Model的桥梁，是对Model的抽象，Model中的数据可以在ViewModel中转换为View所需的格式以供View显示。

• Model

由数据实体、业务对象、数据仓库和服务构成的客户端领域模型，表达数据和业务逻辑。

实现属性或集合的变动通告和数据验证接口。



附录23、 ECMAScript6的部分新特性

- 解构

```
let [x, y] = ['a', 'b'];           // x = 'a'; y = 'b'
let [x, y] = ['a', 'b', 'c'];       // x='a'; y='b';
let [x,...y] = 'abc';               // x='a'; y=['b', 'c'] ...剩余操作符
let [x, y, ...z] = ['a'];           // x='a'; y=undefined; z=[]
let [x, ...[y, z]] = ['a', 'b', 'c']; // x = 'a'; y = 'b'; z = 'c'
let [x] = [];                       // x = undefined
let [x=3, y] = [];                  // x = 3 (默认值); y = undefined
let [x=3, y=x] = [];                // x=3; y=3
let [x=3, y=x] = [7];                // x=7; y=7
let [x=3, y=x] = [7, 2];             // x=7; y=2

let obj = { first: 'Jane', last: 'Doe' };
let { first: f, last: l } = obj;      // f = 'Jane'(obj.first); l = 'Doe'
let obj = { a: [{ foo: 123, bar: 'abc' }, {}], b: true }; // 内嵌
let {length : len} = 'abc';          // len = 3
let {toString: s} = 123;              // s = Number.prototype.toString
let {prop:y} = {};
```

```
let {foo: x=3, bar: y} = {};      // x = 3 (默认值); y = undefined
const FOO = 'foo';
let { [FOO]: f } = { foo: 123 };  // f = 123 []计算表达式作为键值
```

```
let map = new Map().set(false, 'no').set(true, 'yes');
for (let [key, value] of map) {
  console.log(key + ' is ' + value);
}
```

- 可选参数

```
function foo(x=3, y=x) { ... }
foo();           // x=3; y=3
foo(7);          // x=7; y=7
foo(7, 2);       // x=7; y=2
```

- 将 **Set** 转换成数组

```
let set = new Set([11, -1, 6]);
let arr = [...set]; // [11, -1, 6]
```


- 使用**super**

```
class Point {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  toString() {  
    return `${this.x}, ${this.y}`;  
  }  
}  
  
class ColorPoint extends Point {  
  constructor(x, y, color) {  
    super(x, y);  
    this.color = color;  
  }  
  toString() {  
    return super.toString() + ' in ' + this.color;  
  }  
}  
  
> let cp = new ColorPoint(25, 8, 'green');  
> cp.toString();  
'(25, 8) in green'  
> cp instanceof ColorPoint  
true  
> cp instanceof Point  
true
```

- 数组的find和findIndex

返回数组中被回调函数判断为真的第一元素，如果没有该元素，则返回 `undefined`，举例：

```
> [6, -5, 8].find(x => x < 0)
```

```
-5
```

```
> [6, 5, 8].find(x => x < 0)
```

```
undefined
```

`Array.prototype.findIndex(predicate, thisArg?)`返回数组中被回调函数判断为真的第一个元素的索引，如果没有该元素，则返回-1。举例：

```
> [6, -5, 8].findIndex(x => x < 0)
```

```
1
```

```
> [6, 5, 8].findIndex(x => x < 0)
```

```
-1
```

- 数字相关

```
> parseInt('111', 2) // 二进制
```

```
7
```

```
> parseInt('10', 8) // 八进制
```

```
8
```

```
> Math.pow(2, 53)
```

```
9007199254740992
```

- 字符串

// 用\$和反引号实现的字符串字面量的插值

```
let first = 'Jane';
```

```
let last = 'Doe';
```

```
console.log(`Hello ${first} ${last}!`); // Hello Jane Doe!
```

// 字面量的多行字符串

```
let multiLine = `
```

```
This is
```

```
a string
```

```
with multiple
```

```
lines`;
```

“原生”字符串的反斜杠再也不是用于转义的特殊字符，即，`\n` 不会转义成换行符：

```
const str = String.raw`Not a newline: \n`;
```

```
console.log(str === 'Not a newline: \\n'); // true
```

```
> 'hello'.startsWith('hell')
```

```
true
```

```
> 'hello'.endsWith('ello')
```

```
true
```

```
> 'hello'.includes('ell')
```

```
true
```

```
> 'doo '.repeat(3)
```

```
'doo doo doo '
```

每个方法都有可选的第二个参数，用于指定字符串搜索的开始或者结束位置：

```
> 'hello'.startsWith('ello',1)
```

```
true
```

```
> 'hello'.endsWith('hell',4)
```

```
true
```

```
> 'hello'.includes('ell',1)
```

```
true
```

```
> 'hello'.includes('ell',2)
```

```
false
```

- **let和const**

let 创建的变量是可变的:

```
let foo = 'abc';  
foo = 'def';  
console.log(foo); // def
```

常量通过const创建, 是不可变的 - 不能重新赋值:

```
const foo = 'abc';  
foo = 'def'; // TypeError
```

如果const用于定义对象, 对象的属性值是可变的:

```
const obj = {};  
obj.prop = 123;  
console.log(obj.prop); // 123
```

但是不能给obj重新赋值:

```
obj = {}; // TypeError
```

如果能让obj的属性值和方法也不可变, 必须做一些处理, 例如, 冻结对象:

```
const obj = Object.freeze({});  
obj.prop = 123; // TypeError
```

`const` 变量一旦被创建，它就不能被改变。但是这并不意味着你 cannot 通过循环重新进入它的作用域并且赋值重新执行：

```
function logArgs(...args) {  
  for (let [index, elem] of args.entries()) {  
    const message = index + '. ' + elem;  
    console.log(message);  
  }  
}  
logArgs('Hello', 'everyone'); // 0. Hello 1. everyone
```

在以下循环头中允许声明变量：

for
for-in
for-of

如果用`var`，`let`或者`const`声明一个变量，会有不一样的效果。如果使用`let`声明变量，每一次循环都会创建一个`binding`：

```
let arr = [];  
for (let i=0; i < 3; i++) {  
  arr.push(() => i);  
}  
arr.map(x => x()); // [0,1,2] 这里的x为函数，例如:()=>0;
```

每个`i`指向一个循环体中的`binding`并且保持当时的值(闭包)。因此，每个箭头函数都返回不同的值。`const`工作方式与`var`类似，但是不能改变`const`声明变量时初始化的值。刚开始也许会觉得在每个循环中获取新的`binding` 似乎很奇怪，但是当你使用循环创建指向循环变量的函数（例如事件处理的回调）时它非常有用。

- **Symbol**

ES6引入了一种新的原始类型**Symbol**。 **Symbol()**方法返回的每一个**symbol**实例具有唯一ID:

```
> Symbol() === Symbol()  
false
```

```
> typeof Symbol()  
'symbol'
```

Symbol()有一个可选的字符串参数，用作新创建的**Symbol**对象的一个描述文本。该对象的方法**toString()** 或者 **String()**会用到这个参数:

```
> const symb = Symbol('symbol2');  
> String(symb)  
'Symbol(symbol2)'  
> Symbol('hello').toString()  
'Symbol(hello)'
```

Symbol对象可以用作对象属性名和方法名:

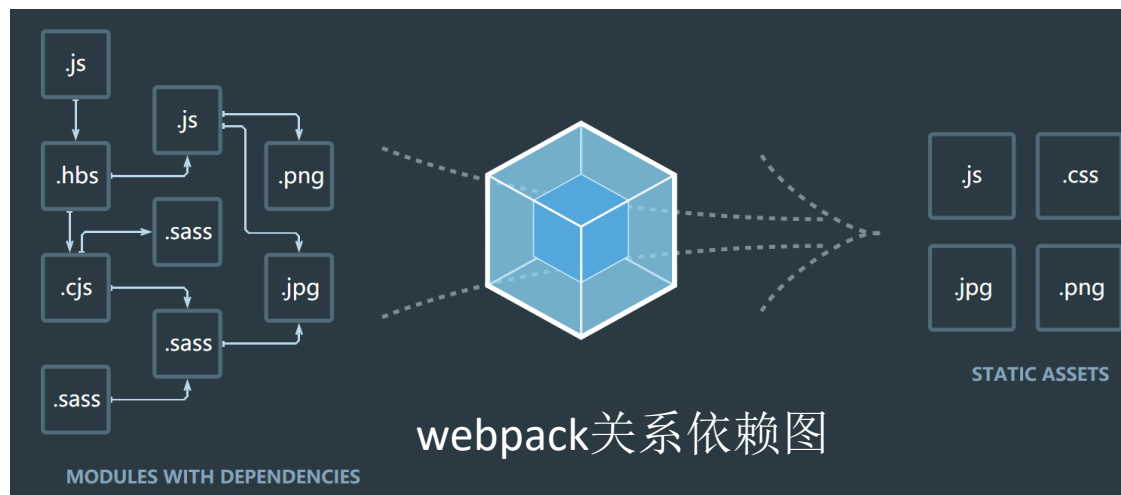
```
const MY_KEY = Symbol();  
const obj = {  
  [MY_KEY]: 123      // []计算表达式作为属性名  
};  
const FOO = Symbol();  
const obj = {        // 作为方法名  
  [FOO]() { return 'bar'; }  
};  
console.log(obj[FOO]());    // bar
```

附录24、babel和webpack

如果希望立即使用ES6或未来的新标准和新提案，但是又怕浏览器不支持，就可以使用babel。babel是一种用途很多的javascript编译器，它可以把最新版的javascript编译成当下可以执行的版本。

转换后的Javascript程序还可以进行压缩，主要是简化命名，用无意义的变量名取代有意义的变量名，然后再用webpack打包。

webpack 是一个 JavaScript程序的静态模块打包器(module bundler)。它会递归地构建一个文件的依赖关系图(dependency graph)，其中包含应用程序需要的每个模块，然后将所有这些模块打包成一个或多个 bundle。



附录25、js常用框架

- React Facebook [教程](#) [教程](#)
 - Vue.js 尤雨溪(Evan You) [教程](#) [教程](#)
 - AngularJS Google
 - Dojo 2
 - Ember
 - Aurelia
 - jquery
 - TypeScript Microsoft
 - Node.js
- [三大框架优缺点](#)

附录26、参考资料

- JavaScript高级程序设计(第3版), (美)Nicholas C.Zakas, 人民邮电出版社 2012年3月
- JavaScript权威指南(第6版), (美)David Flanagan , 机械工业出版社, 2012年5月
- JavaScript面向对象精要, (美)Nicholas C. Zakas, 人民邮电出版社, 2015年4月
- JavaScript语言精粹(修订版), (美)克罗克福德(Crockford,D.), 电子工业出版社 2012年9月
- 精通JavaScript,(美)John Resig等, 人民邮电出版社, 2016年9月
- JavaScript专家编程, [美] Mark Daggett, 人民邮电出版社, 2015 年8月
- JavaScript DOM编程艺术:第2版,(英)jeremy keith;(加)jeffrey sambells(著)人民邮电出版社,2011 年4月
- 移动网页设计与开发HTML5+CSS3+JavaScript,(英)Peter Gasston,人民邮电出版社,2014 年4月
- 原型与闭包 <http://www.cnblogs.com/wangfupeng1988/p/4001284.html>
- 前端开发网址汇总: <https://www.cnblogs.com/i7758/p/5245137.html>