--

[DOM操作基础 1](#_Toc6222)

[children childNodes 1](#_Toc5434)

[JS获取CSS表中的属性 1](#_Toc10957)

[实现拖拽 【事件坐标】 2](#_Toc6085)

[H5的拖动事件 3](#_Toc4024)

[关于页面中的定位 3](#_Toc21371)

[关于页面的高宽 4](#_Toc3280)

[页面总高度 4](#_Toc31910)

[屏幕 4](#_Toc7319)

[对象的高宽 4](#_Toc26008)

[获取对象的高宽（me） 4](#_Toc27274)

[实现广告div随滚轮下降（me） 4](#_Toc27782)

[实现广告div随滚轮下降增强版（me） 4](#_Toc25098)

[关于事件 addEventListener()及 attachEvent()区别分析 4](#_Toc17735)

[JavaScript 事件冒泡简介及应用 5](#_Toc28069)

[默认事件 6](#_Toc26104)

[querySelector和querySelectorAll 6](#_Toc12711)

[其他 7](#_Toc21973)

[关于滚动条的定制 7](#_Toc9230)

[关于focus事件 7](#_Toc22470)

[jquery API熟悉 7](#_Toc13666)

[学习网址 7](#_Toc30325)

[关于事件 7](#_Toc6393)

[.one() 7](#_Toc26083)

[.on()事件绑定传递参数 7](#_Toc28848)

[简单选择器 8](#_Toc3800)

[...closest() 8](#_Toc32550)

[一个参数 8](#_Toc31393)

[两个参数 8](#_Toc23789)

[jQuery object equality 8](#_Toc18941)

[...append() 9](#_Toc20578)

[关于append()的一个注意点 9](#_Toc30756)

[...class 9](#_Toc9467)

[JQ获取对象：类型或值判断、处理 9](#_Toc29515)

[标签类型 9](#_Toc1616)

[其中内容 9](#_Toc7703)

[元素下标 9](#_Toc17453)

[元素个数 9](#_Toc20585)

[对象类型 9](#_Toc30621)

[转化为数组 9](#_Toc3171)

[取第几个 9](#_Toc24093)

[单个重点 9](#_Toc27458)

[jQuery数据缓存 $.data 的使用以及源码解析 9](#_Toc25995)

[www 9](#_Toc22390)

[核心原理 9](#_Toc19860)

[源码总体结构 10](#_Toc32013)

[使用方法 10](#_Toc6356)

[使用方法1 10](#_Toc26582)

[源码 10](#_Toc28928)

[使用方法2 10](#_Toc7011)

[源码 10](#_Toc4421)

[使用方法3 10](#_Toc15473)

[源码 10](#_Toc4263)

[使用方法4 10](#_Toc24129)

[源码 10](#_Toc29810)

[关于prop和attr的区别 10](#_Toc23511)

[使用方式 11](#_Toc29420)

[.prop()和.attr()区别 11](#_Toc9296)

[jquery 源码研究 11](#_Toc20720)

[jquery UI 14](#_Toc23196)

[简介 14](#_Toc22816)

[jquery与jquery UI的插件写法 14](#_Toc27753)

[部件库 widget factory 15](#_Toc8644)

[jQuery UI 通过部件库（Widget Factory）扩展小部件 16](#_Toc8446)

[API 17](#_Toc16026)

DOM操作基础

|  |  |  |
| --- | --- | --- |
| children childNodes | | |
| children  childNodes | **var testdiv**=**document**.getElementById(**"test"**);  **var div1**=**testdiv**.getElementsByTagName(**"div"**);  **var div2**=**testdiv**.**childNodes**;  **var div3**=**testdiv**.**children**;  **console**.log(**div1**.**length**)  **console**.log(**div2**.**length**)  **console**.log(**div3**.**length**)  <**div** id="test">  <**div**></**div**>  <**div**></**div**>  <**div**></**div**>  <**div**></**div**> </**div**> | |
|  | jq:  jQuery.parent(expr) 找父亲节点，可以传入expr进行过滤，比如$("span").parent()或者$("span").parent(".class")  jQuery.parents(expr),类似于jQuery.parents(expr),但是是查找所有祖先元素，不限于父元素  jQuery.children(expr).返回所有子节点，这个方法只会返回直接的孩子节点，不会返回所有的子孙节点  jQuery.contents(),返回下面的所有内容，包括节点和文本。这个方法和children()的区别就在于，包括空白文本，也会被作为一个jQuery对象返回，children()则只会返回节点  jQuery.prev()，返回上一个兄弟节点，不是所有的兄弟节点  jQuery.prevAll()，返回所有之前的兄弟节点  jQuery.next(),返回下一个兄弟节点，不是所有的兄弟节点  jQuery.nextAll()，返回所有之后的兄弟节点  jQuery.siblings(),返回兄弟姐妹节点，不分前后  jQuery.find(expr),跟jQuery.filter(expr)完全不一样。jQuery.filter()是从初始的jQuery对象集合中筛选出一部分，而jQuery.find()的返回结果，不会有初始集合中的内容，比如$("p"),find("span"),是从p元素开始找,等同于$("p span")  js：   |  |  | | --- | --- | | 方法 |  | | appendChild() | 通过把一个节点增加到当前节点的childNodes[]组，给文档树增加节点。 | | cloneNode() | 复制当前节点，或者复制当前节点以及它的所有子孙节点。 | | hasChildNodes() | 如果当前节点拥有子节点，则将返回true。 | | insertBefore() | 给文档树插入一个节点，位置在当前节点的指定子节点之前。如果该节点已经存在，则删除之再插入到它的位置。 | | removeChild() | 从文档树中删除并返回指定的子节点。 | | replaceChild() | 从文档树中删除并返回指定的子节点，用另一个节点替换它。 | | createAttribute() | 用指定的名字创建新的Attr节点。 | | createComment() | 用指定的字符串创建新的Comment节点。 | | createElement() | 用指定的标记名创建新的Element节点。 | | createTextNode() | 用指定的文本创建新的TextNode节点。 | | getElementById() | 返回文档中具有指定id属性的Element节点。 | | getElementsByTagName() | 返回文档中具有指定标记名的所有Element节点。 | | 属性 |  | | attributes | 如果该节点是一个Element，则以NamedNodeMap形式返回该元素的属性。 | | childNodes | 以Node[]的形式存放当前节点的子节点。如果没有子节点，则返回空数组。 | | firstChild | 以Node的形式返回当前节点的第一个子节点。如果没有子节点，则为null。 | | lastChild | 以Node的形式返回当前节点的最后一个子节点。如果没有子节点，则为null。 | | nextSibling | 以Node的形式返回当前节点的兄弟下一个节点。如果没有这样的节点，则返回null。下一个兄弟节点 | | nodeName | 节点的名字，Element节点则代表Element的标记名称。 | | nodeType | 代表节点的类型。 | | parentNode | 以Node的形式返回当前节点的父节点。如果没有父节点，则为null。 | | previousSibling | 以Node的形式返回紧挨当前节点、位于它之前的兄弟节点。如果没有这样的节点，则返回null。上一个兄弟节点 | | |
| JS获取CSS表中的属性 | | |
|  | <div class="indiv"></div> | |
| .indiv{  position:relative;  top:20px;  left:20px;  width:50px;  height:50px;  background:#966;  } | |
| var intt=j(".indiv").css("top");//能取到20px；  var intt=j(".indiv")[0].style.top;//取到的是空字符； | |
|  | var log=function(s){  console.log(s) };  log(document.getElementById("up").style.height);  // 全面版 var GetCurrentStyle=function(obj, prop) {  if (obj.currentStyle) { //IE  return obj.currentStyle[prop];  } else if (window.getComputedStyle) { //非IE /\*大写字母的前面加个‘-’号，并小写\*/  var cssKey = prop.replace(/([A-Z])/g, "-$1").toLowerCase();  return document.defaultView.getComputedStyle(obj, null)[cssKey];  }  return null; }  // 简洁版 var getCss = function(o,key){  return o.currentStyle ?  o.currentStyle[key]:document.defaultView.getComputedStyle(o,false)[key]; };  var up=document.getElementById("up"); log(GetCurrentStyle(up,"width")); log(getCss(up,"width")) | |
|  | ---getComputedStyle是一个可以获取当前元素所有最终使用的CSS属性值。返回的是一个CSS样式声明对象([object CSSStyleDeclaration])，只读。  语法如下：  var style = window.getComputedStyle("元素", "伪类");  例如：  var dom = document.getElementById("test"),  style = window.getComputedStyle(dom , ":after");  例如：  **var style = document**.**defaultView**.getComputedStyle(dom,**false**)["width"]; | |
| 区别 | getComputedStyle方法是只读的，只能获取样式，不能设置；  而element.style能读能写，能屈能伸。  getComputedStyle方法获取的是最终应用在元素上的所有CSS属性对象（即使没有CSS代码，也会把默认的祖宗八代都显示出来）；  而element.style只能获取元素style属性中的CSS样式。  因此对于一个光秃秃的元素<p>，getComputedStyle方法返回对象中length属性值（如果有）就是190+(据我测试FF:192, IE9:195, Chrome:253, 不同环境结果可能有差异), 而element.style就是0。 | |
| 实现拖拽 【事件坐标】 | | |
| JQ实现  拖拽 | $(function(){  $("#xiao").mousedown(function(e){  var $this=$(this);  //事件发生点的坐标（相对于浏览器现）  var x= e.clientX;  var y= e.clientY;  //对象的  var left=$this.position().left;  var top=$this.position().top;  var mousestate=true;   $("#show").html(x+"--"+y+"--"+left+"--"+top);   $(document).mouseover(function(e){  if(mousestate){  $this.css({left:e.clientX-x+left,top: e.clientY-y+top});  }  }).mouseup(function(){  mousestate=false;  });  }); }) | |
| 获取事件  坐标  event  坐标 | event.offsetX 相对对象左上角的水平坐标 event.offsetY 相对对象左上角的垂直坐标  +  event.clientX 相对浏览器左上角的水平座标 event.clientY 相对浏览器左上角的垂直座标 +  event.pageX  event.pageY  event.clientX、event.clientY  鼠标相对于[浏览器](http://www.2cto.com/os/liulanqi/" \t "http://www.2cto.com/kf/201409/_blank)窗口可视区域的X，Y坐标（窗口坐标），可视区域不包括工具栏和滚动条。IE事件和标准事件都定义了这2个属性  Firefox的event不支持offsetX属性,Firefox提供了一个layerX属性，但是该属性需要和CSS的position属性配合才能获取我们想要的数值。而且，layerX的数值严格意义上并不等同于offsetX，offsetX是相对于元素的内边距，而layerX是相对于外边距，这时如果边框较宽，两者取值会有误差，或者说，需要减去边框的宽度。   |  | | --- | | jQuery：  $(e.target).offset().left  原生：  return evt.offsetX || (evt.clientX - srcObj.getBoundingClientRect().left);  function getOffsetX(event){      var evt =event||window.event;      var srcObj = evt.target || evt.srcElement;      if (evt.offsetX){          return evt.offsetX;      }else{          var rect = srcObj.getBoundingClientRect();          var clientx = evt.clientX;          return clientx - rect.left;      }  } | | var box=document.getElementById('box');         // 获取元素  alert(box.getBoundingClientRect().top);         // 元素上边距离页面上边的距离  alert(box.getBoundingClientRect().right);       // 元素右边距离页面左边的距离  alert(box.getBoundingClientRect().bottom);      // 元素下边距离页面上边的距离  alert(box.getBoundingClientRect().left);        // 元素左边距离页面左边的距离  注意：IE、Firefox3+、Opera9.5、Chrome、Safari支持，在IE中，默认坐标从(2,2)开始计算，导致最终距离比其他浏览器多出两个像素，我们需要做个兼容。  document.documentElement.clientTop;  // 非IE为0，IE为2  document.documentElement.clientLeft; // 非IE为0，IE为2  functiongGetRect (element) {      var rect = element.getBoundingClientRect();      var top = document.documentElement.clientTop;      var left= document.documentElement.clientLeft;      return{          top    :   rect.top - top,          bottom :   rect.bottom - top,          left   :   rect.left - left,          right  :   rect.right - left      }  }  分别加上外边据、内边距、边框和滚动条，用于测试所有浏览器是否一致。 |   event.pageX、event.pageY  类似于event.clientX、event.clientY，但它们使用的是文档坐标而非窗口坐标。这2个属性不是标准属性，但得到了广泛支持。IE事件中没有这2个属性。  event.offsetX、event.offsetY  鼠标相对于事件源元素（srcElement）的X,Y坐标，只有IE事件有这2个属性，标准事件没有对应的属性。  event.screenX、event.screenY  鼠标相对于用户显示器屏幕左上角的X,Y坐标。标准事件和IE事件都定义了这2个属性 | |
| H5的拖动事件 | | |
| www | <http://www.zhangxinxu.com/wordpress/2011/02/html5-drag-drop-%E6%8B%96%E6%8B%BD%E4%B8%8E%E6%8B%96%E6%94%BE%E7%AE%80%E4%BB%8B/> | |
|  |  | |
|  | draggable 属性 | 就是标签元素要设置draggable=true，否则不会有效果 |
|  | ondragstart 事件 | 当拖拽元素开始被拖拽的时候触发的事件，此事件作用在被拖曳元素上 |
|  | ondragenter 事件 | 当拖曳元素进入目标元素的时候触发的事件，此事件作用在目标元素上 |
|  | ondragover 事件 | 拖拽元素在目标元素上移动的时候触发的事件，此事件作用在目标元素上 |
|  | ondrop 事件 | 被拖拽的元素在目标元素上同时鼠标放开触发的事件，此事件作用在目标元素上 |
|  | ondragend 事件 | 当拖拽完成后触发的事件，此事件作用在被拖曳元素上 |
|  | e.preventDefault() | 阻止默认的些事件方法等执行。在ondragover中一定要执行preventDefault()，否则ondrop事件不会被触发。另外，如果是从其他应用软件或是文件中拖东西进来，尤其是图片的时候，默认的动作是显示这个图片或是相关信息，并不是真的执行drop。此时需要用用document的ondragover事件把它直接干掉 |
|  | e.effectAllowed | 就是拖拽的效果 |
|  | e.dataTransfer | 退拽对象用来传递的媒介 |
|  | <**style**>  #attrDrop{  position: relative;  }  #attrDrop **div**{  display: inline-block;  vertical-align:top;   }  #div1{  width: 300px;  height: 300px;  border: 3px solid #f08080;  }  #div2{  width: 300px;  height: 300px;  border: 3px solid lightgreen;  } </**style**> <**img** id="drag1" src="../../demo-me/img/face.jpg" draggable="true" ondragstart="drag(event)" /> <**div** id="attrDrop">  <**div** id="div1" ondrop="drop(event)" ondragover="allowDrop(event)">  </**div**>  <**div** id="div2" ondrop="drop(event)" ondragover="allowDrop(event)">  </**div**>  </**div**> <**script** type="text/javascript">  **function allowDrop**(e) {  e.preventDefault();  }  **function drag**(e) {  e.dataTransfer.setData("Text",e.target.id);  }  **function drop**(e) {  e.preventDefault();  **var** data=e.dataTransfer.getData("Text");  e.target.appendChild(document.getElementById(data));  }  **/\*  调用 preventDefault() 来避免浏览器对数据的默认处理（drop 事件的默认行为是以链接形式打开）  通过 dataTransfer.getData("Text") 方法获得被拖的数据。该方法将返回在 setData() 方法中设置为相同类型的任何数据。  被拖数据是被拖元素的 id ("drag1")  把被拖元素追加到放置元素（目标元素）中  \*/** </**script**> | |
| 关于页面中的定位 | | |
| 置顶定位 | 1、直接定位到页面顶部（方式一） <**a** href="#">点击置顶</**a**> \* href="javacript:;"  这种用法不正确，这么用的话会出现浏览器访问"javascript:;"这个地址的现象； \* href="javacript:void(0);"  表示点击这个链接后执行一条javascript语句：void(0); 这条语句表示什么也不做，是个空语句，出现它的原因，就是因为这个链接里面会出现一个点击事件，onclick="sss()"; \* href="#"  这个与Javascript没关系了，这是HTML的链接用法，它并不是没有链接，而是前往当前网页的顶部，如果你的网页比较长，需要向下拉很远的滚动条，这时候你如果想快速地返回到顶部，那么就用这个链接，点一个，网页的滚动条就到最上方了，这种使用#对链接进行标识的用法，叫做锚。 \* 回到顶部的方法： 可以用：  href="###" href="javascript:;" href="javascript:void(0);"  2、缓慢定位到顶部 <**div** id="top">点击置顶</**div**> <**style**>  #top {  position: fixed;  background: lawngreen;  height: 80px;  width: 80px;  bottom: 10px;  right: 10px;  }  #top:hover {  cursor: pointer;  } </**style**> <**script**>  $(**function** () {  //按钮一开始隐藏  //滚动一个屏幕的高度后，按钮显示（client:客户端）  //若再向上滚动  $("#top").css("visibility", "hidden");  **window**.onscroll = **function** () {  **if** (**window**.pageYOffset > **document**.documentElement.clientHeight) {  $("#top").css("visibility", "visible");  } **else** {  $("#top").css("visibility", "hidden");  }  };  //点击之后缓慢向上，在到顶之后取消向上移动  $("#top").click(**function** () {  **var** t = **setInterval**(**function** () {  **var** scroll = **window**.pageYOffset;//专门用来取值[只读]  **document**.body.scrollTop = scroll \* 0.8;//赋值A [可读写360]  **document**.documentElement.scrollTop = scroll \* 0.8;//赋值B [可读写火狐]  **if** (scroll < 1) {  **clearInterval**(t);  }  }, 30)  })  }); </**script**>  3、书签的定位（定位到某个位置） <**a** name="xia" href="#shang">到上面</**a**> ----- <**a** name="shang" href="#xia">到下面</**a**><**br**> | |
| 滚动到  滚动多少  [window] | Css:  html{height:1000px;}  JS： function(){   window.scrollBy(0,100);   alert(window.pageYOffset);  }  Results:  点击一次后弹出：100;整数，只读，默认为0;  附注：  当滚动条到达页面最底部时，window.pageYOffset=1000-document.documentElement.clientHeight;  对浏览器页面整体的操作：  滚动到：window.scrollTo(0,100);  滚动多少：window.scrollBy(0,100); | |
| 滚轮向下  滚动了多少 | window.pageYOffset [只读]  [360有效（能取值，不能赋值，赋值报错）]  [火狐有效（能取值，不能赋值，赋值不报错但无效）]  document.documentElement.scrollTop (360无效)(火狐有效)  document.body.scrollTop (360有效)(火狐无效)  为了兼容，取值可以使用如下代码：  （Y轴上的偏移）（所有元素的滚动高度）（最大元素body的滚动高度）（默认值）  **var scrollTop** = **window**.**pageYOffset** || **document**.**documentElement**.**scrollTop** || **document**.**body**.**scrollTop** || 0; | |
| 关于页面的高宽 | | |
| 页面总高度 | 页面的高度(内在的)（body/html）  （documentElement 对应的是 html 标签，而 body 对应的是 body 标签）  document.body.clientHeight（整个页面的高度）  !=document.documentElement.clientHeight[可见页面的高度，浏览器(客户端)现大小的高度（眼见）]  ==document.body.offsetHeight  ==document.documentElement.offsetHeight[外部]  ==document.body.scrollHeight  ==document.documentElement.scrollHeight[滚动了多少] | |
| 屏幕 | 获取浏览器中显示区域高宽：  **var w** = **window**.**innerWidth**;  **var h** = **window**.**innerHeight**;  **document**.getElementById(**"demo"**).**innerHTML** = **"Width: "** + **w** + **" Heigth: "** + **h**; | |
| 对象的高宽 | scrollHeight: 获取对象的滚动高度。  scrollWidth:获取对象的滚动宽度   scrollLeft:设置或获取位于对象左边界和窗口中目前可见内容的最左端之间的距离 scrollTop:设置或获取位于对象最顶端和窗口中可见内容的最顶端之间的距离  offsetHeight:获取对象相对于版面或由父坐标 offsetParent 属性指定的父坐标的高度  offsetWidth:获取对象 offsetLeft:获取对象相对于版面或由 offsetParent 属性指定的父坐标的计算左侧位置 offsetTop:获取对象相对于版面或由 offsetTop 属性指定的父坐标的计算顶端位置  某个DIV的：clientHeight与height的区别是如果有滚动条时应减去滚动条的17px不可用部分,offsetHeight与Height的区别是增加了boder的高度,ScrollHeihgt与Height的区别是火狐下与offsetHeight一致,IE下如上所述. | |
| 获取对象的高宽（me） | JS.scrollWidth  （小）（实际设置的长度）是对象的实际内容的宽，不包边线宽度，会随对象中内容的多少改变（内容多了可能会改变对象的实际宽度）。  JS.clientWidth  （小）（实际设置的长度）是对象可见的宽度，不包滚动条等边线，会随窗口的显示大小改变。  JS.offsetWidth  （大一点）（实际设置的长度+border）是对象的可见宽度，包滚动条等边线，会随窗口的显示大小改变。 | |
|  | JQ.css("width") （小）（实际设置的长度）  JS.style.width (取不到外部样式设置的值) 可以通过赋值实现修改 | |
| 实现广告div随滚轮下降（me） | <**script**>  $(**function** () {  **setInterval**("**play**()", 2000);  });   // this.offsetTop获得是数值；不可以改变或设置，需通过this.style.top  // this.style.top = document.documentElement.scrolltop+(初始设置的top值)  **function play**() {  $("div").each(**function** () {  **if** (**this**.offsetTop != **document**.documentElement.scrollTop + 80) {  **this**.style.top = **document**.documentElement.scrollTop + 80 + "px";  }  })  } </**script**> | |
| 实现广告div随滚轮下降增强版（me） | **var flag** = **true**;  //绑定点击消失事件，并修改flag值  **j**(**function** () {  **j**(**"img[id\*='close']"**).bind(**"click"**, **function** () {  **flag** = **false**;  **j**(**this**).**parent**().css(**'display'**, **'none'**);  })  })  **function** play2() {  /\*将div的display先设置成none\*/  //alert(screen.width);/\*屏幕分辨率\*/  **if** (**screen**.**width** > 800) {  **if** (**flag**) {  **j**(**"div"**).css(**"display"**, **"block"**);  **j**(**"div"**).each(**function** () {  **if** (**this**.**offsetTop** != **document**.**documentElement**.**scrollTop** + 80) {  **var dx** = (**document**.**documentElement**.**scrollTop** + 80 - **this**.**offsetTop**) \* 0.1;  **dx** = **Math**.*ceil*(**dx**);  **this**.style.**top** = **this**.**offsetTop** + **dx** + **"px"**;  }  })  }  }  } | |

关于事件 addEventListener()及 attachEvent()区别分析

|  |  |
| --- | --- |
|  | Internet Explorer 8 及更早IE版本不支持 addEventListener() 方法，，Opera 7.0 及 Opera 更早版本也不支持。 但是，对于这些不支持该函数的浏览器，你可以使用 attachEvent() 方法来添加事件句柄  **W3C标准中：**  addEventListener的使用方式：  target.addEventListener(type, listener, useCapture);  target： 文档节点、document、window 或 XMLHttpRequest。  type： 字符串，事件名称，不含"on"，比如"click"、"mouseover"、"keydown"等。  listener ：实现了 EventListener 接口或者是 JavaScript 中的函数。  useCapture ：是否使用捕捉，一般用 false 。  **例如：**  **document.getElementById("testText").addEventListener(**  **"keydown", function (event) { alert(event.keyCode); }, false**  **);**   **IE中：**  target.attachEvent(type, listener);  target： 文档节点、document、window 或 XMLHttpRequest。  type： 字符串，事件名称，含"on"，比如"onclick"、"onmouseover"、"onkeydown"等。  listener ：实现了 EventListener 接口或者是 JavaScript 中的函数。  **例如：**  **document.getElementById("txt").attachEvent(**  **"onclick",function(event){alert(event.keyCode);}**  **);**   移除指定的事件, 用途是移除设定的事件, 格式:  **W3C:** removeEventListener(event,function,capture/bubble);  **Windows IE:** detachEvent(event,function); |
|  | addEventListener 用于注册事件处理程序，IE 中为attachEvent，我们为什么讲 addEventListener 而不讲 attachEvent 呢？  一来 attachEvent 比较简单，二来 addEventListener 才是 DOM 中的标准内容。  addEventListener 有三个参数：  **第一个参数表示事件名称（不含 on，如 "click"）；**  **第二个参数表示要接收事件处理的函数；**  **第三个参数为 useCapture，指定事件处理函数的时期或阶段(boolean)。**  true - 事件句柄在捕获阶段执行  false- 默认。事件句柄在冒泡阶段执行  DOM事件流如图（剪自javascript高级程序设计，懒的画了）：    由图可知捕获过程要先于冒泡过程。  当第三个参数设置为true就在捕获过程中执行，反之就在冒泡过程中执行处理函数。 |
| 样例 | outDiv.addEventListener("click", function () { alert(111); }, false); middleDiv.addEventListener("click", function () {alert(111);}, false); inDiv.addEventListener("click", function () {alert(111);}, false);  上述是我们测试的代码，根据 info 的显示来确定触发的顺序，有三个 addEventListener，而 useCapture 可选值为 true 和 false，所以 2\*2\*2，可以得出 8 段不同的程序。   * 全为 false 时，触发顺序为：inDiv、middleDiv、outDiv； * 全为 true 时，触发顺序为：outDiv、middleDiv、inDiv； * outDiv 为 true，其他为 false 时，触发顺序为：outDiv、inDiv、middleDiv； * middleDiv 为 true，其他为 false 时，触发顺序为：middleDiv、inDiv、outDiv； * ……   最终得出如下结论：   * true 的触发顺序总是在 false 之前； * 如果多个均为 true，则外层的触发先于内层；   如果多个均为 false，则内层的触发先于外层。 |
| 实例 | 下面实例讲解如下：  写成这样：只有medhot3被执行  document.getElementById("btn").onclick = method1;  document.getElementById("btn").onclick = method2;  document.getElementById("btn").onclick = method3;  写成这样（IE）:执行顺序为method3->method2->method1  var btn1Obj = document.getElementById("btn1");  //object.attachEvent(event,function);  btn1Obj.attachEvent("onclick",method1);  btn1Obj.attachEvent("onclick",method2);  btn1Obj.attachEvent("onclick",method3);  写成这样（非IE）: 执行顺序为method1->method2->method3  var btn1Obj = document.getElementById("btn1");  //element.addEventListener(type,listener,useCapture);  btn1Obj.addEventListener("click",method1,false);  btn1Obj.addEventListener("click",method2,false);  btn1Obj.addEventListener("click",method3,false); |
| 公共方法-事件绑定 | **if** (**x**.addEventListener) {  **x**.addEventListener(**"click"**, myFunction);  } **else if** (**x**.attachEvent) {  // IE 8 及更早 IE 版本  **x**.attachEvent(**"onclick"**, myFunction);  } |
|  | **function addEV**(d, b, a) {  **if** (**window**.attachEvent) {  d.attachEvent("on" + b, a)  } **else** {  **if** (**window**.addEventListener) {  d.addEventListener(b, a, **false**)  }  } } |
| 公共方法-事件绑定 | /\*关于事件的绑定2\*/ // 注意代码中的注释部分。因为在符合DOM标准的浏览器中，addEventListener方法将把this指针指向绑定的函数，而IE中attachEvent方法将始终指向widow对象，为了将this指针始终指向当前绑定事件的DOM，我们必须要使用apply或者call方法来改变函数的作用域。 **function** bind(el, type, fn, useCapture) {  **if** (**window**.addEventListener) {  el.addEventListener(type, **function** () {  fn.apply(el, **arguments**);//始终将this指向DOM  }, useCapture);  } **else if** (**window**.attachEvent) {  el.attachEvent(**'on'** + type, **function** () {  fn.apply(el, **arguments**);//始终将this指向DOM  });  } }  **var el** = **document**.getElementById(**'addEventListener'**); **var** test = **function** () {  alert(**this**.nodeName); }; bind(**el**, **'click'**, test); |
|  |  |

JavaScript 事件冒泡简介及应用

|  |  |  |
| --- | --- | --- |
|  | 一、什么是事件冒泡 在一个对象上触发某类事件（比如单击onclick事件），如果此对象定义了此事件的处理程序，那么此事件就会调用这个处理程序，如果没有定义此事件处理程序或者事件返回true，那么这个事件会向这个对象的父级对象传播，从里到外，直至它被处理（父级对象所有同类事件都将被激活），或者它到达了对象层次的最顶层，即document对象（有些浏览器是window）。   打个比方说：你在地方法院要上诉一件案子，如果地方没有处理此类案件的法院，地方相关部门会帮你继续往上级法院上诉，比如从市级到省级，直至到中央法院，最终使你的案件得以处理。   二、事件冒泡有什么作用  （1）事件冒泡允许多个操作被集中处理（把事件处理器添加到一个父级元素上，避免把事件处理器添加到多个子级元素上），它还可以让你在对象层的不同级别捕获事件。  【集中处理例子】  <**style**>  #outSide {  width: 100px;  height: 100px;  background: #000;  padding: 50px;  }  #inSide {  width: 100px;  height: 100px;  background: #CCC;  }  </**style**> <**div** onclick="**eventHandle**(**event**)" id="outSide">  <**div** id="inSide"></**div**> </**div**> <**script** type="text/javascript">  /\*\*  \* 本例子只在外面盒子定义了处理方法，而这个方法一样可以捕获到子元素点击行为并处理它。假设有成千上万子元素要处理，难道我们要为每个元素加onclick="eventHandle(event)"？  \* 显然没有这种集中处理的方法来的简单，同时它的性能也是更高的。  \* 点击inside区域 inside is click  \* 点击outside区域 outside is click  \* \*/  **function eventHandle**(e) {  **var** e = e || **window**.event;  **var** obj = e.target || e.srcElement;  **console**.log(obj.id + ' was click');  } </**script**>  （2）让不同的对象同时捕获同一事件，并调用自己的专属处理程序做自己的事情，就像老板一下命令，各自员工做自己岗位上的工作去了。   【同时捕获同一事件例子】  <**div** onclick="**outSideWork**()" id="outSide">  <**div** onclick="**inSideWork**()" id="inSide"></**div**> </**div**>  三、需要注意什么  ●事件捕获其实有三种方式，事件冒泡只是其中的一种：（1）IE从里到外（inside→outside）的冒泡型事件。（2）Netscape4.0从外到里（outside→inside）的捕获型事件。（3）DOM事件流，先从外到里，再从里到外回到原点（outside→inside→outside）的事件捕获方法（似乎对象将触发两次事件处理，这有什么作用？鄙人不懂！）。   ●不是所有的事件都能冒泡。以下事件不冒泡：blur、focus、load、unload。   ●事件捕获方式在不同浏览器，甚至同种浏览器的不同版本中是有所区别的。如Netscape4.0采用捕获型事件解决方案，其它多数浏览器则支持冒泡型事件解决方案，另外DOM事件流还支持文本节点事件冒泡。   ●事件捕获到达顶层的目标在不同浏览器或不同浏览器版本也是有区别的。在IE6中HTML是接收事件冒泡的，另外大部分浏览器将冒泡延续到window对象，即……body→documen→window。   ●阻止冒泡并不能阻止对象默认行为。比如submit按钮被点击后会提交表单数据，这种行为无须我们写程序定制。   四、阻止事件冒泡  通常情况下我们都是一步到位，明确自己的事件触发源，并不希望浏览器自作聪明、漫无目的地去帮我们找合适的事件处理程序，即我们明确最精准目标，这种情况下我们不需要事件冒泡。另外通过对事件冒泡的理解，我们知道程序将做多较多额外的事情，这必然增大程序开销。还有一个重要的问题是：事件冒泡处理可能会激活我们本来不想激活的事件，导致程序错乱，甚至无从下手调试，这常成为对事件冒泡不熟悉程序员的棘手问题。所以必要时，我们要阻止事件冒泡。   【阻止事件冒泡例子】  **<!--  点击inside区域  冒泡：1 0  阻止冒泡：1 -->** <**div** onclick="**show**(0)" id="outSide"  style="width:100px; height:100px; background:#000; padding:50px">  <**div** onclick="**show**(1)" id="inSide"  style="width:100px; height:100px; background:#CCC"></**div**> </**div**> <**script** type="text/javascript">  **function show**(value, e) {  **if** (e && e.stopPropagation) {  e.stopPropagation()  } **else** {  **window**.event.cancelBubble = **true** }  **console**.log(value);  } </**script**> | |
| 默认事件 | | | |
| 阻止默认事件 | | event.preventDefault()方法是用于取消事件的默认行为，例如，当点击提交按钮时阻止对表单的提交。但此方法并不被ie支持，在ie下需要用window.event.returnValue = false; 来实现。  1、如果 Event 对象的 cancelable 属性是 fasle，那么就没有默认动作，或者不能阻止默认动作。无论哪种情况，调用该方法都没有作用。  2、此方法并不被ie支持，在ie下需要用window.event.returnValue = false; 来实现。    <**script** type="text/javascript">  **function stopDefault**(e) {  **if** (e && e.preventDefault) {//如果是FF下执行这个  e.preventDefault();  } **else** {  **window**.event.returnValue = **false**;//如果是IE下执行这个  }  **return false**;  } </**script**> <**a** href="url" id="test">测试</**a**> <**script** type="text/javascript">  **document**.getElementById('test').onclick = **function** (e) {  **alert**('URL：' + **this**.href + ', 不会跳转');  **stopDefault**(e);  } </**script**> | |
| 总结 | | 阻止默认事件行为的触发:  event.preventDefault();  window.event.returnValue=false;  防止事件冒泡到DOM树上，也就是不触发的任何前辈元素上的事件处理函数:  event.stopPropagation();  window.event.cancelBubble=true; | |

[querySelector和querySelectorAll](http://www.cnblogs.com/dolphinX/p/3354318.html)

|  |  |
| --- | --- |
|  | 著作权归作者所有。 商业转载请联系作者获得授权，非商业转载请注明出处。 作者：简生 链接：http://www.zhihu.com/question/24702250/answer/28695133 来源：知乎  **1. W3C 标准** querySelectorAll 属于 W3C 中的 Selectors API 规范 [1]。而 getElementsBy 系列则属于 W3C 的 DOM 规范 [2]。  **2. 浏览器兼容** querySelectorAll 已被 IE 8+、FF 3.5+、Safari 3.1+、Chrome 和 Opera 10+ 良好支持 。 getElementsBy 系列，以最迟添加到规范中的 getElementsByClassName 为例，IE 9+、FF 3 +、Safari 3.1+、Chrome 和 Opera 9+ 都已经支持该方法了。  **3. 接收参数** querySelectorAll 方法接收的参数是一个 CSS 选择符。而 getElementsBy 系列接收的参数只能是单一的className、tagName 和 name。代码如下 [3]：  var c1 = document.querySelectorAll('.b1 .c'); var c2 = document.getElementsByClassName('c'); var c3 = document.getElementsByClassName('b2')[0].getElementsByClassName('c');  需要注意的是，querySelectorAll 所接收的参数是必须严格符合 CSS 选择符规范的。所以下面这种写法，将会抛出异常。代码如下 [4]：  <script>  try {  var e1 = document.getElementsByClassName('1a2b3c');  var e2 = document.querySelectorAll('.1a2b3c');  }catch (e) {  console.error(e.message);  }  console.log(e1 && e1[0].className);  console.log(e2 && e2[0].className); </script>  （CSS 选择器中的元素名，类和 ID 均不能以数字为开头。）  **4. 返回值** 大部分人都知道，querySelectorAll 返回的是一个 Static Node List，而 getElementsBy 系列的返回的是一个 Live Node List。 看看下面这个经典的例子 [5]：  // Demo 1 动态的 var ul = document.getElementsByTagName('ul')[0],  lis = ul.getElementsByTagName("li"); for (var i = 0; i < lis.length; i++) { //死循环  alert(lis.length)  ul.appendChild(document.createElement("li")); } // Demo 2 静态的 var ul = document.querySelectorAll('ul')[0],  lis = ul.querySelectorAll("li"); for (var i = 0; i < lis.length; i++) {  alert(lis.length)  ul.appendChild(document.createElement("li")); }  因为 Demo 1 中的 lis 是一个动态的 Node List， 每一次调用 lis 都会重新对文档进行查询，导致无限循环的问题。 而 Demo 2 中的 lis 是一个静态的 Node List，是一个 li 集合的快照，对文档的任何操作都不会对其产生影响。  但为什么要这样设计呢？ 其实，在 W3C 规范中对 querySelectorAll 方法有明确规定 [6]：  The NodeList object returned by the querySelectorAll() method **must be static** ([DOM], section 8).  那什么是 NodeList 呢？ W3C 中是这样说明的 [7]：  The NodeList interface provides the abstraction of an ordered collection of nodes, without defining or constraining how this collection is implemented. **NodeList objects in the DOM are live.**  所以，NodeList 本质上是一个动态的 Node 集合，只是规范中对 querySelectorAll 有明确要求，规定其必须返回一个静态的 NodeList 对象。  我们再看看在 Chrome 上面是个什么样的情况：  document.querySelectorAll('a').toString();  // return "[object NodeList]" document.getElementsByTagName('a').toString();  // return "[object HTMLCollection]"  这里又多了一个 HTMLCollection 对象出来，那 HTMLCollection 又是什么？  HTMLCollection 在 W3C 的定义如下 [8]：  An HTMLCollection is a list of nodes. An individual node may be accessed by either ordinal index or the node's name or id attributes. Note: Collections in the HTML DOM are assumed to be live meaning that they are automatically updated when the underlying document is changed.  实际上，HTMLCollection 和 NodeList 十分相似，都是一个动态的元素集合，每次访问都需要重新对文档进行查询。两者的本质上差别在于，HTMLCollection 是属于 **Document Object Model HTML** 规范，而 NodeList 属于 **Document Object Model Core** 规范。 这样说有点难理解，看看下面的例子会比较好理解 [9]：  <**ul**>  <**li**>仄仄仄仄仄</**li**>  <**li**>仄仄仄仄仄</**li**>  <**li**>仄仄仄仄仄</**li**>  <**li**>仄仄仄仄仄</**li**> </**ul**> <**script** type="text/javascript">  **var ul** = **document**.getElementsByTagName(**'ul'**)[0],  **lis1** = **ul**.**childNodes**,  **lis2** = **ul**.children;  **console**.log(**lis1**.toString(), **lis1**.**length**); // "[object NodeList]" 9  **console**.log(**lis2**.toString(), **lis2**.**length**); //"[object HTMLCollection]" 4 </**script**>  NodeList 对象会包含文档中的所有节点，如 Element、Text 和 Comment 等。 HTMLCollection 对象只会包含文档中的 Element 节点。  另外，HTMLCollection 对象比 NodeList 对象 多提供了一个 namedItem 方法。  所以在现代浏览器中，querySelectorAll 的返回值是一个静态的 NodeList 对象，而 getElementsBy 系列的返回值实际上是一个 HTMLCollection 对象 。 |
|  | 1. querySelector：   return the first matching Element node within the node’s subtrees. If there is no such node, the method must return null .  （返回指定元素节点的子树中匹配选择器的集合中的第一个元素，如果没有匹配返回null）   1. querySelectorAll：   return a NodeList containing all of the matching Element nodes within the node’s subtrees, in document order. If there are no such nodes, the method must return an empty NodeList.  （按文档顺序返回指定元素节点的子树中匹配选择器的元素集合，如果没有匹配返回空集合）  从定义可以看到Document和Element都实现了NodeSelector接口。即这三种类型的元素都拥有者两个方法。querySelector和querySelectorAll的参数是CSS选择器字符串。区别在于querySelector返回的是一个第一个匹配元素，querySelectorAll返回的一个所有匹配元素集合(NodeList)。 |
|  | <div id="test">  <div></div>  <div></div>  <div></div> </div> <div class="myclass">  <div> </div>  <div> </div>  <div name="a"> </div> </div>  var test=document.querySelector('#test'); //undefined var subDivs = test.querySelectorAll('div'); //3 var myclass = document.querySelectorAll('div[class=myclass]'); //1 var myclass2 = document.querySelectorAll('.myclass'); //1 var myclass3 = document.querySelectorAll('.myclass div'); //3 var myclass4 = document.querySelectorAll('.myclass div[name=a]'); //1 |

其他

|  |  |
| --- | --- |
| 关于滚动条的定制 | 通过css实现  <http://www.lyblog.net/detail/314.html>  关于JS滚轮事件mousewheel、DOMMouseScroll  <http://www.zhangxinxu.com/wordpress/2013/04/js-mousewheel-dommousescroll-event/> |
| 关于focus事件 | **<!-- 通过hidefocus style 可以控制focus的样式 style="border:1px solid #ccc;width:200px;height:200px;outline:0;" hidefocus="false" -->** <**div** tabindex="0" onfocus='**console**.log(**"得到焦点"**);' onblur='**console**.log(**"失去焦点"**);' onkeydown="xKeyEvent(**event**)">sdfasdf</**div**>  <**script**>  **function** xKeyEvent(e) { //通用的兼容各个浏览器的响应键盘事件函数//FUNCTION: do the EVENT of KEY  **var e** = e || **window**.**event**; //事件处理  **console**.log(**"111111111"**)  //按键 ASCII 码值  **console**.log(**e**.**keyCode** | **e**.**which** | **e**.**charCode**)   **switch** (**e**.**keyCode** | **e**.**which** | **e**.**charCode**) {  //响应退格键  **case** 13:{  **console**.log(**"Enter::::"**)  //...x operation...  **break**;  }  }  } </**script**> |
|  |  |
|  |  |

jquery API熟悉

|  |  |  |
| --- | --- | --- |
| 学习网址 | jQuery API 中文文档  <http://www.jquery123.com/>  jQuery API 英文文档  <http://api.jquery.com/>  网页制作辅助工具-jQuery标尺参考线插件  <http://www.zhangxinxu.com/wordpress/2010/08/%E7%BD%91%E9%A1%B5%E5%88%B6%E4%BD%9C%E8%BE%85%E5%8A%A9%E5%B7%A5%E5%85%B7-jquery%E6%A0%87%E5%B0%BA%E5%8F%82%E8%80%83%E7%BA%BF%E6%8F%92%E4%BB%B6/>  jQuery powerFloat万能浮动层下拉层插件  <http://www.zhangxinxu.com/wordpress/2010/12/jquery-powerfloat%E4%B8%87%E8%83%BD%E6%B5%AE%E5%8A%A8%E5%B1%82%E4%B8%8B%E6%8B%89%E5%B1%82%E6%8F%92%E4%BB%B6/>  可想造一个属于你自己的jQuery库?(九，实现on&off)  <http://div.io/topic/1472>  实现图片的异步加载  <http://div.io/topic/1769> | |
| 关于事件 | | |
| jquery | e.preventDefault()  e.isDefaultPrevented() | 关于浏览器的默认事件 |
|  | e.target | $("p, button, h1, h2").click(function(event){  $("div").html(event.target.nodeName);  }); |
| e.pageX  e.pageY | $(document).mousemove(function(e){  $("span").text("X: " + e.pageX + ", Y: " + e.pageY);  }); |
| e.which | $("input").keydown(function(event){  $("div").html("Key: " + event.which);  });  哪个按键被按 |
| mouse | mousedown mouseup 鼠标移动到元素上方并按下鼠标按钮[左右点击都有效]  mouseover mouseout 父子元素都有效[单一][过程：上方飘过]  mouseenter mouseleave 只对父元素有效[单一][过程：进入]  mousemove  <http://www.w3school.com.cn/tiy/t.asp?f=jquery_event_mouseenter_mouseover> |
|  |  |
|  |  |
|  |  |
| .one() | 被绑定的事件只会运行一次 |
| .on()事件绑定传递参数 | **function** myHandler(event) {  alert(event.data.foo);  }  $(**"p"**).on(**"click"**, {**foo**: **"bar"**}, myHandler) |
|  |  |
|  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### 简单选择器

|  |  |  |
| --- | --- | --- |
| j("li:first") | j("li").first() |  |
| j("li:last") | j("li").last() |  |
| j("li:eq(0)") | j("li").eq(0) |  |
| j("input:not(:checked)") | j("li").not($("#se")[0]) |  |
| j("li:gt(0)") | j("li").slice(1) |  |
| j("li:lt(5)") | j("li").slice[0,5) |  |
| j("li:even") |  | begin from zero!!!(0) |
| j("li:odd") |  | begin from zero!!!(0) |
| j("li:not(:animated)")  .animate({ left: "+=20" }, 1000); |  |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |
| j("li:") | j("li"). |  |

...closest()

|  |  |
| --- | --- |
| 一个参数 | <**ul** id="one" class="level-1">  <**li** class="item-i">I</**li**>  <**li** id="ii" class="item-ii">II  <**ul** class="level-2">  <**li** class="item-a">A</**li**>  <**li** class="item-b">B  <**ul** class="level-3">  <**li** class="item-1">1</**li**>  <**li** class="item-2">2</**li**>  <**li** class="item-3">3</**li**>  </**ul**>  </**li**>  <**li** class="item-c">C</**li**>  </**ul**>  </**li**>  <**li** class="item-iii">III</**li**> </**ul**> |
| $(**'li.item-1'**).closest(**'li'**).css(**'background-color'**, **'red'**);  ---离自身(包括自己)开始最近的某某元素， |
|  |
| 两个参数 | $(**'li.item-1'**).closest(**'ul'**).css(**'background-color'**, **'red'**); |
|  |
| **var listItemII** = **document**.getElementById(**'ii'**);  $(**'li.item-1'**).closest(**'#one'**, **listItemII**).css(**'background-color'**, **'green'**);  $(**'li.item-1'**).closest(**'#one'**).css(**'background-color'**, **'green'**);  ---第二个参数限制范围，在第二参数所定义的元素之中未找到则返回空 |
|  |

jQuery object equality

|  |  |
| --- | --- |
|  | http://stackoverflow.com/questions/3176962/jquery-object-equality |
|  | How do I determine if two jQuery objects are equal? I would like to be able to search an array for a particular jQuery object.  $.inArray(jqobj, my\_array);//-1  alert($("#deviceTypeRoot") == $("#deviceTypeRoot"));//False  alert($("#deviceTypeRoot") === $("#deviceTypeRoot"));//False |
|  | Since jQuery 1.6, you can use [.is](http://api.jquery.com/is/). Below is the answer from over a year ago...  var a = $('#foo');  var b = a;  if (a.is(b)) {  // the same object!  }  If you want to see if two variables are actually the same object, eg:  var a = $('#foo');  var b = a;  ...then you can check their unique IDs. Every time you create a new jQuery object it gets an id.  if ($.data(a) == $.data(b)) {  // the same object!  }  Though, the same could be achieved with a simple a === b, the above might at least show the next developer exactly what you're testing for.  In any case, that's probably not what you're after. If you wanted to check if two different jQuery objects contain the same set of elements, the you could use this:  $.fn.equals = function(compareTo) {  if (!compareTo || this.length != compareTo.length) {  return false;  }  for (var i = 0; i < this.length; ++i) {  if (this[i] !== compareTo[i]) {  return false;  }  }  return true;  };  [Source](http://groups.google.com/group/jquery-dev/browse_thread/thread/86504e9660e893ef?pli=1)  var a = $('p');  var b = $('p');  if (a.equals(b)) {  // same set  } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

...append()

|  |  |
| --- | --- |
| 关于append()的一个注意点 | **<div id="text"></div>**  **<div>**  **<ul>**  **<li>one</li>**  **<li>two</li>**  **<li>three1</li>**  **<li>four1</li>**  **</ul>**  **</div>** |
| **var j=jQuery.noConflict();**  **j(function(){**  **var a=j("li").slice(0,2).toArray();**  **for(var i=0;i<a.length;i++){**  **j("#text").append(a[i]);**  **}**  **})**  **// 自动截取后面的** |
| **j("#text").append(a[i].innerHTML);**  **// 保留后面的** |

### ...class

|  |  |  |
| --- | --- | --- |
| 关于class的判断及处理 | addClass("a b"):向此元素添加一个或多个类，中间用空格隔开  j("sth").addClass("red");  j("sth").addClass("red bigger");  removeClass():无参数将所有的class删除，不会删除id选择器  j("sth").removeClass().addClass("red");  j("sth").removeClass("red");  hasClass(): 检查被选元素是否包含指定的 class,返回"false""true"。  不加参数，暂定为总返回"false"，故无此用法。  j("sth").hasClass("red");  toggleClass():该方法检查每个元素中指定的类。如果不存在则添加类，如果已设置则删除之。这就是所谓的切换效果。不过，通过使用 "switch" 参数，您能够规定只删除或只添加类。  ===$(selector).toggleClass(class,switch)  P1:必需。规定添加或移除 class 的指定元素；如需规定若干 class，请使用空格来分隔类名。  P2:可选。布尔值。规定是否添加或移除 class。  ===$(selector).toggleClass(function(index,class),switch)  P1:必需。规定返回需要添加或删除的一个或多个类名的函数。index - 可选。接受选择器的 index 位置。class - 可选。接受选择器的当前的类。  P2:可选。布尔值。规定是否添加(true)或移除(false)类。  ===Demo:  $(e.target).closest('.btn').toggleClass('focus', /^focus(in)?$/.test(e.type)) | |
| 响应事件实现变色 | j("button").click(function(){  j("div").toggleClass("red");  })  实现点击变色 |  |
| j("button").click(function(){  j("li").toggleClass(function(){  return "li-"+j(this).index()%2;  })  })  })  高级的点击实现隔行变色。 | <**style**>  .li-0{  color:#F66;  }  .li-1{  color:#00F;  } </**style**> |

JQ获取对象：类型或值判断、处理

|  |  |  |
| --- | --- | --- |
| 标签类型 | j("sth").get(0).nodeName;  J("sth")[0].nodename; |  |
| 其中内容 | j("sth").html() |  |
| 元素下标 | j("sth").index() |  |
| 元素个数 | j("sth").size()  j("sth").length |  |
| 对象类型 | typeof 123;  j.type(123); |  |
| 转化为数组 | j.type(j("li").toArray())  j.type(j("li"))  j("li").toArray().length  j("li").length | Array  Object  4  4 | |
| 取第几个 | j("sth").get(0)  j("sth")[0]  j("sth").slice(0,1)  j("sth").eq(0)  J("sth").first() | js obj  js obj  jq obj  jq obj  jq obj | |
|  |  |  | |
|  |  |  | |
|  |  |  | |
|  |  |  | |

单个重点

|  |  |
| --- | --- |
| [jQuery数据缓存 $.data 的使用以及源码解析](https://segmentfault.com/a/1190000000626031) | |
|  |  |
| www | https://segmentfault.com/a/1190000000626031 |
| 核心原理 | 对于DOM元素：  通过分配一个唯一的关联id把DOM元素和该DOM元素的数据缓存对象关联起来，关联id被附加到以jQuery.expando的值命名的属性上，数据存储在全局缓存对象jQuery.cache中。在读取、设置、移除数据时，将通过关联id从全局缓存对象jQuery.cache中找到关联的数据缓存对象，然后在数据缓存对象上执行读取、设置、移除操作。  对于Javascript对象：  数据则直接存储在该Javascript对象的属性jQuery.expando上。在读取、设置、移除数据时，实际上是对Javascript对象的数据缓存对象执行读取、设置、移除操作。  为了避免jQuery内部使用的数据和用户自定义的数据发生冲突，数据缓存模块把内部数据存储在数据缓存对象上，把自定义数据存储在数据缓存对象的属性data上。 |
| 源码总体结构 | // 数据缓存 Data **jQuery**.extend({  // 全局缓存对象  **cache**: {},  // 唯一 id种子  **uuid**:0,  // 页面中每个jQuery副本的唯一标识  **expando**: **"jQuery"** + ( **jQuery**.**fn**.**jquery** + **Math**.*random*() ).replace( /\D/g, **""** ),  // 是否有关联的数据  hasData: **function**(){},  // 设置、读取自定数据或内部数据  data: **function**(elem, name, data, pvt) {},  // 移除自定义数据或内部数据  removeData: **function**(elem, name, pvt) {},  // 设置、读取内部数据  \_data: **function**(elem, name, data) {},  // 是否可以设置数据  acceptData: **function**(elem){} }); **jQuery**.**fn**.extend({  // 设置、读取自定义数据，解析HTML5属性data-  data: **function**(key,value){},  // 移除自定义数据  removeData: **function**(key){} }); // 解析HTML5属性 data- **function** dataAttr(elem,key,data){} // 检查数据缓存对象是否为空 **function** isEmptyDataObject(obj){} **jQuery**.extend({  // 清空数据缓存对象  cleanData: **function**(elems){} }); |
| 使用方法 | $.data(elem, name, data), $.data(elem, name)  $.data(key, value), $.data(key)  $.removeData(elem, name)，$.removeData(key) |
| 使用方法1 | 1. 放值：$.data(elem, name, data) 2. 取值：$.data(elem, name)   **var div** = $(**"div"**)[0]; $.data(**div**, **"test"**, {**first**: 16, **last**: **"pizza!"**}); $.data(**div**, **"test"**).first; $.data(**div**, **"test"**).last; |
| 源码 | jQuery.extend({  // 1. 定义jQuery.data(elem, name, data, pvt)  data: **function**( elem, name, data, pvt /\* Internal Use Only \*/ ) {  // 2. 检查是否可以设置数据  **if** ( !jQuery.acceptData( elem ) ) {  **return**; // 如果参数elem不支持设置数据，则立即返回  }   // 3 定义局部变量  **var privateCache**, **thisCache**, **ret**,  **internalKey** = jQuery.**expando**,  **getByName** = **typeof** name === **"string"**,   // We have to handle DOM nodes and JS objects differently because IE6-7  // can't GC object references properly across the DOM-JS boundary  **isNode** = elem.**nodeType**, // elem是否是DOM元素   // Only DOM nodes need the global jQuery cache; JS object data is  // attached directly to the object so GC can occur automatically  **cache** = **isNode** ? jQuery.**cache** : elem,  // 如果是DOM元素，为了避免javascript和DOM元素之间循环引用导致的浏览器(IE6/7)垃圾回收机制不起作用，要把数据存储在全局缓存对象jQuery.cache中；对于javascript对象，来及回收机制能够自动发生，不会有内存泄露的问题，因此数据可以查收存储在javascript对象上   // Only defining an ID for JS objects if its cache already exists allows  // the code to shortcut on the same path as a DOM node with no cache  **id** = **isNode** ? elem[ **internalKey** ] : elem[ **internalKey** ] && **internalKey**,  **isEvents** = name === **"events"**;   // Avoid doing any more work than we need to when trying to get data on an  // object that has no data at all  // 4. 如果是读取数据，但没有数据，则返回  **if** ( (!**id** || !**cache**[**id**] || (!**isEvents** && !pvt && !**cache**[**id**].data)) && **getByName** && data === **undefined** ) {  **return**;  // getByName && data === undefined 如果name是字符串，data是undefined, 说明是在读取数据  // !id || !cache[id] || (!isEvents && !pvt && !cache[id].data 如果关联id不存在，说明没有数据；如果cache[id]不存在，也说明没有数据；如果是读取自动以数据，但cache[id].data不存在，说明没有自定义数据  }   // 5. 如果关联id不存在，则分配一个  **if** ( !**id** ) {  // Only DOM nodes need a new unique ID for each element since their data  // ends up in the global cache  **if** ( **isNode** ) {  elem[ **internalKey** ] = **id** = ++jQuery.**uuid**; // 对于DOM元素，jQuery.uuid会自动加1，并附加到DOM元素上  } **else** {  **id** = **internalKey**; // 对于javascript对象，关联id就是jQuery.expando  }  }   // 6. 如果数据缓存对象不存在，则初始化为空对象{}  **if** ( !**cache**[ **id** ] ) {  **cache**[ **id** ] = {};   // Avoids exposing jQuery metadata on plain JS objects when the object  // is serialized using JSON.stringify  **if** ( !**isNode** ) {  **cache**[ **id** ].**toJSON** = jQuery.noop; // 对于javascript对象，设置方法toJSON为空函数，以避免在执行JSON.stringify()时暴露缓存数据。如果一个对象定义了方法toJSON(),JSON.stringify()在序列化该对象时会调用这个方法来生成该对象的JSON元素  }  }   // An object can be passed to jQuery.data instead of a key/value pair; this gets  // shallow copied over onto the existing cache  // 7. 如果参数name是对象或函数，则批量设置数据  **if** ( **typeof** name === **"object"** || **typeof** name === **"function"** ) {  **if** ( pvt ) {  **cache**[ **id** ] = jQuery.extend( **cache**[ **id** ], name ); // 对于内部数据，把参数name中的属性合并到cache[id]中  } **else** {  **cache**[ **id** ].**data** = jQuery.extend( **cache**[ **id** ].data, name ); // 对于自定义数据，把参数name中的属性合并到cache[id].data中  }  }   // 8. 如果参数data不是undefined, 则设置单个数据  **privateCache** = **thisCache** = **cache**[ **id** ];   // jQuery data() is stored in a separate object inside the object's internal data  // cache in order to avoid key collisions between internal data and user-defined  // data.  **if** ( !pvt ) {  **if** ( !**thisCache**.data ) {  **thisCache**.**data** = {};  }   **thisCache** = **thisCache**.data;  }   **if** ( data !== **undefined** ) {  **thisCache**[ jQuery.camelCase( name ) ] = data;  }   // Users should not attempt to inspect the internal events object using jQuery.data,  // it is undocumented and subject to change. But does anyone listen? No.  // 9. 特殊处理events   **if** ( **isEvents** && !**thisCache**[ name ] ) { // 如果参数name是字符串"events",并且未设置过自定义数据"events",则返回事件婚车对象，在其中存储了事件监听函数。  **return privateCache**.**events**;  }   // Check for both converted-to-camel and non-converted data property names  // If a data property was specified  //10. 如果参数name是字符串，则读取单个数据  **if** ( **getByName** ) {   // First Try to find as-is property data  **ret** = **thisCache**[ name ]; // 先尝试读取参数name对应的数据   // Test for null|undefined property data  **if** ( **ret** == **null** ) { // 如果未取到，则把参数name转换为驼峰式再次尝试读取对应的数据   // Try to find the camelCased property  **ret** = **thisCache**[ jQuery.camelCase( name ) ];  }  } **else** { // 11. 如果未传入参数name,data,则返回数据缓存对象  **ret** = **thisCache**;  }   **return ret**;  },   // For internal use only.  \_data: **function**( elem, name, data ) {  **return** jQuery.data( elem, name, data, **true** );  }, }); |
| 使用方法2 | 1. 放值：$("str").data(key,value) 2. 取值：$("str").data(key)   $(**"body"**).data(**"foo"**, 52 ); // 放值 $(**"body"**).data(**"bar"**, { **myType**: **"test"**, **count**: 40 } ); // 放值 $(**"body"**).data(**"foo"** ); // 52 // 取值 $(**"body"**).data({**baz**:[1,2,3]}); // ？？？ $(**"body"**).data(); // ？？？  HTML5 data attriubutes:  <div data-role="page"  data-last-value="43"  data-hidden="true"  data-options='{"name":"John"}'>  </div>  $("div").data("role") === "page";  $("div").data("lastValue") === 43;  $("div").data("hidden") === true;  $("div").data("options").name === "John"; |
| 源码 | jQuery.fn.extend({  // 1. 定义.data(key, value)  data: function( key, value ) {  var parts, attr, name,  data = null;  // 2. 未传入参数的情况  if ( typeof key === "undefined" ) {  if ( this.length ) { // 如果参数key是undefined, 即参数格式是.data(), 则调用方法jQuery.data(elem, name, data, pvt)获取第一个匹配元素关联的自定义数据缓存对象，并返回。  data = jQuery.data( this[0] );  if ( this[0].nodeType === 1 && !jQuery.\_data( this[0], "parsedAttrs" ) ) {  attr = this[0].attributes;  for ( var i = 0, l = attr.length; i < l; i++ ) {  name = attr[i].name;  if ( name.indexOf( "data-" ) === 0 ) {  name = jQuery.camelCase( name.substring(5) );  dataAttr( this[0], name, data[ name ] );  }  }  jQuery.\_data( this[0], "parsedAttrs", true );  }  }  return data;  // 3. 参数key 是对象的情况,即参数格式是.data(key),则遍历匹配元素集合，为每个匹配元素调用方法jQuery.data(elem, name, data,pvt)批量设置数据  } else if ( typeof key === "object" ) {  return this.each(function() {  jQuery.data( this, key );  });  }  // 4. 只传入参数key的情况 如果只传入参数key, 即参数格式是.data(key),则返回第一个匹配元素的指定名称数据  parts = key.split(".");  parts[1] = parts[1] ? "." + parts[1] : "";  if ( value === undefined ) {  data = this.triggerHandler("getData" + parts[1] + "!", [parts[0]]);  // Try to fetch any internally stored data first  if ( data === undefined && this.length ) {  data = jQuery.data( this[0], key );  data = dataAttr( this[0], key, data );  }  return data === undefined && parts[1] ?  this.data( parts[0] ) :  data;  // 5. 传入参数key和value的情况 即参数格式是.data(key, value),则为每个匹配元素设置任意类型的数据，并触发自定义事件setData, changeData  } else {  return this.each(function() {  var self = jQuery( this ),  args = [ parts[0], value ];  self.triggerHandler( "setData" + parts[1] + "!", args );  jQuery.data( this, key, value );  self.triggerHandler( "changeData" + parts[1] + "!", args );  });  }  },  removeData: function( key ) {  return this.each(function() {  jQuery.removeData( this, key );  });  }  });  // 6. 函数dataAttr(elem, key, data)解析HTML5属性data-  function dataAttr( elem, key, data ) {  // If nothing was found internally, try to fetch any  // data from the HTML5 data-\* attribute  // 只有参数data为undefined时，才会解析HTML5属性data-  if ( data === undefined && elem.nodeType === 1 ) {  var name = "data-" + key.replace( rmultiDash, "-$1" ).toLowerCase();  data = elem.getAttribute( name );  if ( typeof data === "string" ) {  try {  data = data === "true" ? true :  data === "false" ? false :  data === "null" ? null :  jQuery.isNumeric( data ) ? parseFloat( data ) :  rbrace.test( data ) ? jQuery.parseJSON( data ) :  data;  } catch( e ) {}  // Make sure we set the data so it isn't changed later  jQuery.data( elem, key, data );  } else {  data = undefined;  }  }  return data;  } |
| 使用方法3 | $.removeData(elem, name)  $("").removeData(key)  var div = $( "div" )[ 0 ];  $( "span:eq(0)" ).text( "" + $( "div" ).data( "test1" ) ); //undefined  jQuery.data( div, "test1", "VALUE-1" );  jQuery.data( div, "test2", "VALUE-2" );  $( "span:eq(1)" ).text( "" + jQuery.data( div, "test1" ) ); // VALUE-1  jQuery.removeData( div, "test1" );  $( "span:eq(2)" ).text( "" + jQuery.data( div, "test1" ) ); // undefined  $( "span:eq(3)" ).text( "" + jQuery.data( div, "test2" ) ); // value2  $( "span:eq(0)" ).text( "" + $( "div" ).data( "test1" ) ); // undefined  $( "div" ).data( "test1", "VALUE-1" );  $( "div" ).data( "test2", "VALUE-2" );  $( "span:eq(1)" ).text( "" + $( "div").data( "test1" ) ); // VALUE-1  $( "div" ).removeData( "test1" );  $( "span:eq(2)" ).text( "" + $( "div" ).data( "test1" ) ); // undefined  $( "span:eq(3)" ).text( "" + $( "div" ).data( "test2" ) ); // VALUE-2 |
| 源码 | $.extend({  // jQuery.removeData(elem,name,pvt)用于移除通过jQuery.data()设置的数据  removeData: function( elem, name, pvt /\* Internal Use Only \*/ ) {  if ( !jQuery.acceptData( elem ) ) {  return;  }  var thisCache, i, l,  // Reference to internal data cache key  internalKey = jQuery.expando,  isNode = elem.nodeType,  // See jQuery.data for more information  cache = isNode ? jQuery.cache : elem,  // See jQuery.data for more information  id = isNode ? elem[ internalKey ] : internalKey;  // If there is already no cache entry for this object, there is no  // purpose in continuing  if ( !cache[ id ] ) {  return;  }  // 如果传入参数name, 则移除一个或多个数据  if ( name ) {  thisCache = pvt ? cache[ id ] : cache[ id ].data;  if ( thisCache ) { // 只有数据缓存对象thisCache存在时，才有必要移除数据  // Support array or space separated string names for data keys  if ( !jQuery.isArray( name ) ) {  // try the string as a key before any manipulation  if ( name in thisCache ) {  name = [ name ];  } else {  // split the camel cased version by spaces unless a key with the spaces exists  name = jQuery.camelCase( name );  if ( name in thisCache ) {  name = [ name ];  } else {  name = name.split( " " );  }  }  }  // 遍历参数name中的数据名，用运算符delete逐个从数据缓存对象thisCache中移除  for ( i = 0, l = name.length; i < l; i++ ) {  delete thisCache[ name[i] ];  }  // If there is no data left in the cache, we want to continue  // and let the cache object itself get destroyed  if ( !( pvt ? isEmptyDataObject : jQuery.isEmptyObject )( thisCache ) ) {  return;  }  }  }  // See jQuery.data for more information  // 删除自定义数据缓存对象cache[id].data  if ( !pvt ) {  delete cache[ id ].data;  // Don't destroy the parent cache unless the internal data object  // had been the only thing left in it  if ( !isEmptyDataObject(cache[ id ]) ) {  return;  }  }  // Browsers that fail expando deletion also refuse to delete expandos on  // the window, but it will allow it on all other JS objects; other browsers  // don't care  // Ensure that `cache` is not a window object #10080  // 删除数据缓存对象cache[id]  if ( jQuery.support.deleteExpando || !cache.setInterval ) {  delete cache[ id ];  } else {  cache[ id ] = null;  }  // We destroyed the cache and need to eliminate the expando on the node to avoid  // false lookups in the cache for entries that no longer exist  // 删除DOM元素上扩展的jQuery.expando属性  if ( isNode ) {  // IE does not allow us to delete expando properties from nodes,  // nor does it have a removeAttribute function on Document nodes;  // we must handle all of these cases  if ( jQuery.support.deleteExpando ) {  delete elem[ internalKey ];  } else if ( elem.removeAttribute ) {  elem.removeAttribute( internalKey );  } else {  elem[ internalKey ] = null;  }  }  }  });  jQuery.fn.extend({  removeData: function( key ) {  return this.each(function() {  jQuery.removeData( this, key );  });  }  });  // checks a cache object for emptiness  function isEmptyDataObject( obj ) {  for ( var name in obj ) {  // if the public data object is empty, the private is still empty  if ( name === "data" && jQuery.isEmptyObject( obj[name] ) ) {  continue;  }  if ( name !== "toJSON" ) {  return false;  }  }  return true;  } |
| 使用方法4 | $.hasData(elem)  检测元素是否拥有与之相关的任何 jQuery 数据  P1：可选：需要检查其数据的DOM元素  XX：  jQuery.hasData() 方法检测元素当前是否拥有通过使用 jQuery.data() 设置的任何值。如果没有数据与元素相关（根本不存在数据对象或者数据对象为空），则该方法返回 false；否则返回 true。  jQuery.hasData(element) 的主要优势是，在不存在数据对象的情况下，不会创建并将数据对象与元素进行关联。相反地，jQuery.data(element) 总是向调用者返回数据对象，如果之前数据对象不存在，则会创建它。  请注意，jQuery的事件系统是使用jQuery数据 存储事件处理程序的。 因此，使用.on(), .bind(), .live(), .delegate()，或一个速记事件方法 绑定事件到一个元素上的时候，也会在那个元素上关联一个 data 对象。  var $p = jQuery( "p" ), p = $p[ 0 ];  $p.append( jQuery.hasData( p ) + " " ); // false  $.data( p, "testing", 123 );  $p.append( jQuery.hasData( p ) + " " ); // true  $.removeData( p, "testing" );  $p.append( jQuery.hasData( p ) + " " ); // false  $p.on( "click", function() {} );  $p.append( jQuery.hasData( p ) + " " ); // true  $p.off( "click" );  $p.append( jQuery.hasData( p ) + " " ); // false |
| 源码 | $.extend({  hasData: function( elem ) {  elem = elem.nodeType ? jQuery.cache[ elem[jQuery.expando] ] : elem[ jQuery.expando ];  return !!elem && !isEmptyDataObject( elem );  // 如果关联的数据缓存对象存在，并且含有数据，则返回true, 否则返回false。 这里用两个逻辑非运算符! 把变量elem转换为布尔值  }  }); |

关于prop和attr的区别

|  |  |
| --- | --- |
| 使用方式 | prop() 方法设置或返回被选元素的属性和值。  .prop( propertyName )  获取匹配集合中第一个元素的Property的值  .prop( propertyName, value )  .prop( map )  .prop( propertyName, function(index, oldPropertyValue) )  给匹配元素集合设定一个或多个属性  当该方法用于返回属性值时，则返回第一个匹配元素的值。  当该方法用于设置属性值时，则为匹配元素集合设置一个或多个属性/值对。   1. 注意：   prop() 方法应该用于检索属性值，例如 DOM 属性（如 selectedIndex, tagName, nodeName, nodeType, ownerDocument, defaultChecked, 和 defaultSelected）。  提示：如需检索 HTML 属性，请使用 [attr()](http://www.runoob.com/jquery/html-attr.html) 方法代替。  提示：如需移除属性，请使用 [removeProp()](http://www.runoob.com/jquery/html-removeprop.html) 方法。  根据官方的建议：具有 true 和 false 两个属性的属性，如 checked, selected 或者 disabled 使用prop()，其他的使用 attr() |
| .prop()和.attr()区别 | 下面是关于jQuery1.6和1.6.1中Attributes模块变化的描述，以及.attr()方法和.prop()方法的首选使用  Attributes模块的变化是移除了attributes和properties之间模棱两可的东西，但是在jQuery社区中引起了一些混乱，因为在1.6之前的所有版本中都使用一个方法(.attr())来处理attributes和properties。但是老的.attr()方法有一些bug，很难维护。jQuery1.6.1对Attributes模块进行了更新，并且修复了几个bug。   |  |  |  | | --- | --- | --- | | elem.checked | true(Boolean) | Will change with checkbox state | | $(elem).prop("checked") | true(Boolean) | Will change with checkbox state | | elem.getAttribute("checked") | "checked"(String) | Initial state of the checkbox;does not change | | $(elem).attr("checked")(1.6) | "checked"(String) | Initial state of the checkbox;does not change | | $(elem).attr("checked")(1.6.1+) | "checked"(String) | Will change with checkbox state | | $(elem).attr("checked")(pre-1.6) | true(Boolean) | Changed with checkbox state | | if(elem.checked)  if($(elem).prop("checked"))  if($(elem).is(":checked"))  这三个都是返回Boolean值。 | | |   为了让jQuery1.6中的.attr()方法的变化被理解的清楚些，下面是一些使用.attr()的例子，虽然在jQuery之前的版本中能正常工作，但是现在必须使用.prop（）方法代替：  IMG_256  首先，window或document中使用.attr()方法在jQuery1.6中不能正常运行，因为window和document中不能有attributes。它们包含properties(比如：location或readyState)，必须使用.prop()方法操作或简单地使用javascript原生的方法。在jQuery1.6.1中，window和document中使用.attr()将被自动转成使用.prop，而不是抛出一个错误。  其次，checked,selected和前面提到的其它boolean attributes，因为这些attributes和其相应的properties之间的特殊关系而被特殊对待。基本上，一个attribute就是以下html中你看到的：  <input type=”checkbox”checked=”checked”>  boolean attributes，比如：checked，仅被设置成默认值或初始值。在一个checkbox的元素中，checked attributes在页面加载的时候就被设置，而不管checkbox元素是否被选中。  properties就是浏览器用来记录当前值的东西。正常情况下，properties反映它们相应的attributes(如果存在的话)。但这并不是boolean attriubutes的情况。当用户点击一个checkbox元素或选中一个select元素的一个option时，boolean properties保持最新。但相应的boolean attributes是不一样的，正如上面所述，它们仅被浏览器用来保存初始值。  $(“:checkbox”).get(0).checked=true;  //Is the same as$(":checkbox:first").prop(“checked”,true);  在jQuery1.6中，如果使用下面的方法设置checked：  $(“:checkbox”).attr(“checked”,true);  将不会检查checkbox元素，因为它是需要被设置的property，但是你所有的设置都是初始值。  然而，曾经jQuery1.6被释放出来的时候，jQuery团队明白当浏览器仅关心页面加载时，设置一些值不是特别的有用。所以，为了保持向后兼容性和.attr()方法的有用性，我们可以继续在jQuery1.6.1中使用.attr()方法取得和设置这些boolean attributes。  最普通的attributes是checked,selected,disabled和readOnly，但下面是jQuery1.6.1支持的使用.attr()动态地取得和设置boolean attributes/properties的完整列表：  autofocus,autoplay,async,checked,controls,defer,disabled,hidden,loop,multiple,open,readonly,required,scoped,selected  还是建议使用.prop()方法来设置这些boolean attributes/properties，即使这些用例没有转换成使用.prop()方法，但是你的代码仍然可以在jQuery1.6.1中正常运行。  下面是一些attributes和properties的列表，正常情况下，应该使用其对应的方法(见下面的列表)来取得和设置它们。下面的是首用法，但是.attr()方法可以运行在所有的attributes情况下。  注意：一些DOM元素的properties也被列在下面，但是仅运行在新的.prop()方法中  IMG_256 IMG_256  \*例如:window.location  \*\*如果需要在(if needed over).width()  .attr()和.prop()都不应该被用来取值/设值。使用.val()方法代替(即使使用.attr("value","somevalue")可以继续运行，就像1.6之前做的那样)  3、首选用法的概述  .prop()方法应该被用来处理boolean attributes/properties以及在html(比如：window.location)中不存在的properties。其他所有的attributes(在html中你看到的那些)可以而且应该继续使用.attr()方法来进行操作。  上面的概述已经描述的够清楚了，我也没有必要再总结了。 |
|  |  |

## jquery 源码研究

|  |  |
| --- | --- |
| www | jquery 学习网址——较好的一个教程集  <http://www.365mini.com/doc>  <http://www.365mini.com/page/jquery-quickstart.htm> |
| 前言 | 从整体入手理解框架结构，逐渐攻破每个依赖模块  [回调函数][异步加载][数据缓存][动画队列]  [选择器][DOM处理][事件][ajax][动画模块] |
|  | The Write Less,Do More（写更少，做更多），无疑就是jQuery的核心理念，简洁的API、优雅的链式、强大的查询与便捷的操作。从而把jQuery打造成前端世界的一把利剑，所向披靡！  简洁的API:  $.on  $.css  $.ajax  ….  优雅的链式:  var jqxhr = $.ajax( "example.php" )  .done(function() { alert("success"); })  .fail(function() { alert("error"); })  .always(function() { alert("complete"); });  强大的选择器：  $("div, span, p.myClass" )  $("div span:first-child")  $("tr:visible")  …  便捷的操作：  $("p").removeClass("myClass noClass").addClass("yourClass");  $("ul li:last").addClass(function(index) {  return"item-" + index;  });  $('.container').append($('h2'));  …  为什么要做jQuery源码解析？  虽然jQuery的文档很完善，潜意识降低了前端开发的入门的门槛，要实现一个动画随手拈来，只要简单的调用一个animate方法传递几个执行的参数即可，但如果要我们自己实现一个定制的动画呢？我们要考虑的问题太多太多了，浏览器兼容、各种属性的获取、逻辑流程、性能等等，这些才是前端开发的基础核心。  如果我们只知道使用jQuery，而不知道其原理，那就是“知其然,而不知其所以然”，说了这么多，那就赶快跟着慕课网进入“高大上”之旅吧，深入来探究jQuery的内部架构！ |
|  | ;(function(global, factory) {  factory(global);  }(typeof window !== "undefined" ? window : this, function(window, noGlobal) {  var jQuery = function( selector, context ) {  return new jQuery.fn.init( selector, context );  };  jQuery.fn = jQuery.prototype = {};  // 核心方法  // 回调系统  // 异步队列  // 数据缓存  // 队列操作  // 选择器引  // 属性操作  // 节点遍历  // 文档处理  // 样式操作  // 属性操作  // 事件体系  // AJAX交互  // 动画引擎  return jQuery;  })); |
|  |  |
|  | jQuery一共13个模块，从2.1版开始jQuery支持通过AMD模块划分，jQuery在最开始发布的1.0版本是很简单的，只有CSS选择符、事件处理和AJAX交互3大块。其发展过程中，有几次重要的变革：  ☑ 1.2.3 版发布，引入数据缓存，解决循环引用与大数据保存的问题  ☑ 1.3 版发布，它使用了全新的选择符引擎Sizzle，在各个浏览器下全面超越其他同类型JavaScript框架的查询速度，程序库的性能也因此有了极大提升  ☑ 1.5 版发布，新增延缓对像(Deferred Objects)，并用deferred重写了Ajax模块  ☑ 1.7 版发布，抽象出回调对象，提供了强大的的方式来管理回调函数列表。 |
|  | jQuery按我的理解分为五大块，选择器、DOM操作、事件、AJAX与动画， |
| ajax | jQuery文档针对业务层的Ajax的处理提供了一系列的门面接口：  .ajaxComplete()  .ajaxError()  .ajaxSend()  .ajaxStart()  .ajaxStop()  .ajaxSuccess()  底层接口：  jQuery.ajax()  jQuery.ajaxSetup()  快捷方法：  jQuery.get()  jQuery.getJSON()  jQuery.getScript()  jQuery.post()  我们在深入内部看看Ajax的高层方法其实都是统一调用了一个静态的jQuery.ajax方法：  jQuery.each( [ "get", "post" ], function( i, method ) {  jQuery[ method ] = function( url, data, callback, type ) {  // Shift arguments if data argument was omitted  if ( jQuery.isFunction( data ) ) {  type = type || callback;  callback = data;  data = undefined;  }  return jQuery.ajax({  url: url,  type: method,  dataType: type,  data: data,  success: callback  });  };  });  在jQuery.ajax的内部实现是非常复杂的，首先ajax要考虑异步的处理与回调的统一性，所以就引入了异步队列模块（Deferred）与回调模块（Callbacks）, 所以要把这些模块方法在ajax方法内部再次封装成、构建出一个新的jQXHR对象，针对参数的默认处理，数据传输的格式化等等。 |
| 立即调用表达式 | 任何库与框架设计的第一个要点就是解决命名空间与变量污染的问题。jQuery就是利用了JavaScript函数作用域的特性，采用立即调用表达式包裹了自身的方法来解决这个问题。  jQuery的立即调用函数表达式的写法有三种：  写法1：  (function(window, factory) {  factory(window)  }(this, function() {  return function() {  //jQuery的调用  }  }))  可以看出上面的代码中嵌套了2个函数，而且把一个函数作为参数传递到另一个函数中并且执行，这种方法有点复杂，我们简化一下写法：  写法2：  var factory = function(){  return function(){  //执行方法  }  }  var jQuery = factory();  上面的代码效果和方法1是等同的，但是这个factory有点变成了简单的工厂方法模式，需要自己调用，不像是一个单例的jQuery类，所以我们需要改成“自执行”，而不是另外调用。  写法3：  (function(window, undefined) {  var jQuery = function() {}  // ...  window.jQuery = window.$ = jQuery;  })(window);  从上面的代码可看出，自动初始化这个函数，让其只构建一次。详细说一下这种写法的优势：    1、window和undefined都是为了减少变量查找所经过的scope作用域。当window通过传递给闭包内部之后，在闭包内部使用它的时候，可以把它当成一个局部变量，显然比原先在window scope下查找的时候要快一些。   2、undefined也是同样的道理，其实这个undefined并不是JavaScript数据类型的undefined，而是一个普普通通的变量名。只是因为没给它传递值，它的值就是undefined，undefined并不是JavaScript的保留字。    有童鞋留言到，为什么要传递undefined？  [Java](http://www.2cto.com/kf/ware/Java/" \t "http://www.imooc.com/code/_blank)script 中的 undefined 并不是作为关键字，因此可以允许用户对其赋值。  我们看一个  var undefined = '慕课网'  ;(function(window) {  alert(undefined);//IE8 '慕课网'  })(window)  IE8存在这个问题，当然，大部分浏览器都是不能被修改的  如果函数调用不传递，参数默认就是undefined  ;(function(window,undefined) {  //undefined  })(window)    jQuery为什么要创建这样的一个外层包裹，其原理又是如何？  这里要区分2个概念一个是匿名函数，一个是自执行。顾名思义，匿名函数，就是没有函数名的函数，也就是不存在外部引用。但是是否像下面代码实现呢：  function(){  //代码逻辑  }  上面这种写法是错了，声明了它但是又不给名字又没有使用，所以在语法上错误的，那么怎么去执行一个匿名的函数呢？ 要调用一个函数，我们必须要有方法定位它、引用它。所以，我们要取一个名字：  var jQuery = function(){  //代码逻辑  }  jQuery使用()将匿名函数括起来，然后后面再加一对小括号（包含参数列表），那么这小括号能把我们的表达式组合分块，并且每一块（也就是每一对小括号），都有一个返回值。这个返回值实际上也就是小括号中表达式的返回值。所以，当我们用一对小括号把匿名函数括起来的时候，实际上小括号返回的，就是一个匿名函数的Function对象。因此，小括号对加上匿名函数就如同有名字的函数般被我们取得它的引用位置了。所以如果在这个引用变量后面再加上参数列表，就会实现普通函数的调用形式。  最后，我们回到写法1看看jQuery利用写法3的写法，然后把整个函数作为参数传递给另外一个函数，主要是为了判断jQuery在不同平台的下的加载逻辑，主流的库一般都有对 AMD 和 CommonJS 的支持代码，看看jQuery的代码：  if (typeof module === "object" && typeof module.exports === "object") {  module.exports = global.document ?  factory(global, true) :  function(w) {  if (!w.document) {  throw new Error("jQuery requires a window with a document");  }  return factory(w);  };  } else {  factory(global);  }  总结：全局变量是魔鬼, 匿名函数可以有效的保证在页面上写入JavaScript，而不会造成全局变量的污染，通过小括号，让其加载的时候立即初始化，这样就形成了一个单例模式的效果从而只会执行一次。 |
| jQuer的类数组对象结构 | 首先我们看jQuery的入口都是统一的$, 通过传递参数的不同，实现了9种方法的重载：  1. jQuery([selector,[context]])  2. jQuery(element)  3. jQuery(elementArray)  4. jQuery(object)  5. jQuery(jQuery object)  6. jQuery(html,[ownerDocument])  7. jQuery(html,[attributes])  8. jQuery()  9. jQuery(callback)  9种用法整体来说可以分三大块：选择器、dom的处理、dom加载。  换句话说jQuery就是为了获取DOM、操作DOM而存在的！所以为了更方便这些操作，让节点与实例对象通过一个桥梁给关联起来，jQuery内部就采用了一种叫“类数组对象”的方式作为存储结构，所以我们即可以像对象一样处理jQuery操作，也能像数组一样可以使用push、pop、shift、unshift、sort、each、map等类数组的方法操作jQuery对象了。  jQuery对象可用数组下标索引是什么原理？  通过$(".Class")构建的对象结构如下所示：  [IMG_256](http://img.mukewang.com/53fad4240001c7b805050236.jpg)                     整个结构很明了，通过对象键值对的关系保存着属性，原型保存着方法。  函数aQuery()内部首先保证了必须是通过new操作符构建。这样就能保证当前构建的是一个带有this的实例对象，既然是对象我们可以把所有的属性与方法作为对象的key与value的方式给映射到this上，所以如上结构就可以模拟出jQuery的这样的操作了，即可通过索引取值，也可以链式方法取值，但是这样的结构是有很大的缺陷的，每次调用ajQuery方法等于是创建了一个新的实例，那么类似get方法就要在每一个实例上重新创建一遍，性能就大打折扣，所以jQuery在结构上的优化不仅仅只是我们看到的，除了实现类数组结构、方法的原型共享，而且还实现方法的静态与实例的共存，这是我们之后将会重点分析的。  var aQuery = function(selector) {  //强制为对象  if (!(this instanceof aQuery)) {  return new aQuery(selector);  }  var elem = document.getElementById(/[^#].\*/.exec(selector)[0]);  this.length = 1;  this[0] = elem;  this.context = document;  this.selector = selector;  this.get = function(num) {  return this[num];  }  return this;  } |
| ready与load事件 | jQuery有3种针对文档加载的方法  $(document).ready(function() {  // ...代码...  })  //document ready 简写  $(function() {  // ...代码...  })  $(document).load(function() {  // ...代码...  })  ready先执行，load后执行。  DOM文档加载的步骤： 要想理解为什么ready先执行，load后执行就要先了解下DOM文档加载的步骤：  (1) 解析HTML结构。  (2) 加载外部脚本和样式表文件。  (3) 解析并执行脚本代码。  (4) 构造HTML DOM模型。//ready  (5) 加载图片等外部文件。  (6) 页面加载完毕。//load  ready与load的区别就在于资源文件的加载，ready构建了基本的DOM结构，所以对于代码来说应该越快加载越好。在一个高速浏览的时代，没人愿意等待答案。假如一个网站页面加载超过4秒，不好意思，你1/4的用户将面临着流失，所以对于框架来说用户体验是至关重要的，我们应该越早处理DOM越好，我们不需要等到图片资源都加载后才去处理框架的加载，图片资源过多load事件就会迟迟不会触发。  jQuery是如何处理文档加载时机的问题：  jQuery.ready.promise = function( obj ) {  if ( !readyList ) {  readyList = jQuery.Deferred();  if ( document.readyState === "complete" ) {  // Handle it asynchronously to allow scripts the opportunity to delay ready  setTimeout( jQuery.ready );  } else {  document.addEventListener( "DOMContentLoaded", completed, false );  window.addEventListener( "load", completed, false );  }  }  return readyList.promise( obj );  };  jQuery的ready是通过promise给包装过的，这也是jQuery擅长的手法，统一了回调体系，以后我们会重点谈到。 可见jQuery兼容的具体策略：针对高级的浏览器，我们当前很乐意用DOMContentLoaded事件了，省时省力。 |
|  | show('观察脚本加载的顺序')  document.addEventListener("DOMContentLoaded", function() {  show('DOMContentLoaded回调')  }, false);  window.addEventListener("load", function() {  show('load事件回调')  }, false);  show('脚本解析一')  //测试加载  $(function(){  show('脚本解析二')  })  show('脚本解析三') |
| jQuery多库共存处理 | 多库共存换句话说可以叫无冲突处理。  总的来说会有2种情况会遇到：     1、$太火热，jQuery采用$作为命名空间，不免会与别的库框架或者插件相冲突。     2、jQuery版本更新太快，插件跟不上，导致不同版本对插件的支持度不一样。  出于以上的原因，jQuery给出了解决方案–– noConflict函数。  使用DEMO：  jQuery.noConflict();  // 使用 jQuery  jQuery("aaron").show();  // 使用其他库的 $()  $("aaron").style.display = ‘block’;  这个函数必须在你导入jQuery文件之后，并且在导入另一个导致冲突的库之前使用。当然也应当在其他冲突的库被使用之前，除非jQuery是最后一个导入的。 |
|  | Var \_jQuery = window.jQuery,  \_$ = window.$;  jQuery.noConflict = function( deep ) {  if ( window.$ === jQuery ) {  window.$ = \_$;  }  if ( deep && window.jQuery === jQuery ) {  window.jQuery = \_jQuery;  }  return jQuery;  };      如果我们需要同时使用jQuery和其他JavaScript库，我们可以使用 $.noConflict()把$的控制权交给其他库。旧引用的$ 被保存在jQuery的初始化; noConflict() 简单的恢复它们。     通过类似swap交换的概念，先把之前的存在的命名空间给缓存起来，通过对比当前的命名空间达到交换的目的，首先，我们先判断下当前的的$空间是不是被jQuery接管了，如果是则让出控制权给之前的\_$引用的库，如果传入deep为true的话等于是把jQuery的控制权也让出去了。     如果不通过noConflict处理的话其后果可想而知，香喷喷的$大家都“觊觎已久”。 |
|  | $("#aaron").click(function() {  $.noConflict(); //让出控制权  if (!$) {  show("使用noConflict后，$不存在")  }  if (jQuery) {  show("使用noConflict后，jQuery存在")  }  //通过闭包隔离出$  ;(function($) {  if ($) {  show("通过闭包隔离后，转为局部变量$存在")  }  })(jQuery);  }) |
| 对象的构建 | //类一：  function ajQuery() {  this.name = 'jQuery';  this.sayName = function() {  return this.name  }  }  var a = new ajQuery()  var b = new ajQuery()  var c = new ajQuery()      // 类二  function ajQuery() {  this.name = 'jQuery'  }  ajQuery.prototype = {  sayName: function() {  return this.name  }  }  var a = new ajQuery()  var b = new ajQuery()  var c = new ajQuery()  // jQuery的写法  var $jQuery = function(selector, context) {  return new $jQuery.fn.init(selector, context);  }  $jQuery.fn = $jQuery.prototype = {  init: function() {  this.name = 'aaron'  return this;  },  constructor: $jQuery,  -------  }  var $a = $jQuery();  show($a); |
|  | 类一与类二产生的结构几乎是一样的，而本质区别就是：类二new产生的a、b、c三个实例对象共享了原型的sayName方法，这样的好处节省了内存空间，类一则是要为每一个实例复制sayName方法，每个方法属性都占用一定的内存的空间，所以如果把所有属性方法都声明在构造函数中，就会无形的增大很多开销，这些实例化的对象的属性一模一样，都是对this的引用来处理。除此之外类一的所有方法都是拷贝到当前实例对象上。类二则是要通过scope连接到原型链上查找，这样就无形之中要多一层作用域链的查找了。  jQuery对象的构建如果在性能上考虑，所以就必须采用原型式的结构。  使用原型结构，性能上是得到了优化，但是ajQuery类这个结构与目标jQuery的结构的还是有很大不一致：     ☑   没有采用new操作符；     ☑   return返回的是一个通过new出来的的对象 。 |

## jquery UI

简介

|  |  |  |  |
| --- | --- | --- | --- |
| 特点 | 简单易用  继承 jQuery 简易使用特性，提供高度抽象接口，短期改善网站易用性。  开源免费  采用 MIT & GPL 双协议授权，轻松满足自由产品至企业产品各种授权需求。  广泛兼容  兼容各主流桌面浏览器。包括IE 6+、Firefox 2+、Safari 3+、Opera 9+、Chrome 1+。  轻便快捷  组件间相对独立，可按需加载，避免浪费带宽拖慢网页打开速度。  标准先进  支持 WAI-ARIA，通过标准 XHTML 代码提供渐进增强，保证低端环境可访问性。  美观多变  提供近 20 种预设主题，并可自定义多达 60 项可配置样式规则，提供 24 种背景纹理选择。 | | |
|  | 缺点、不足  1.代码不够健壮：缺乏全面的测试用例，部分组件 Bugs 较多，不能达到企业级产品开发要求。  2.构架规划不足：组件间 API 缺乏协调，缺乏配合使用帮助。  3.控件较少：相对于 Dojo、YUI、Ext JS 等成熟产品，可用控件较少，无法满足复杂界面功能要求。 | | |
| 工作原理 |  | | |
| jquery与jquery UI的插件写法 | | | |
| jquery | | $.fn.plugin=function(options){  options=$.extend({},$.fn.plugin.default,options);  }  $.fn.plugin.default={  param1:1,  param2:2,  param3:3  } | |
| jquery UI | | $.widget("ns.plugin",{  options:{ },  \_create:function(){ }  }) | |
|  | | [jQuery](http://caibaojian.com/jquery/" \o "jQuery手册" \t "http://blog.csdn.net/yummy_go/article/details/_blank)为开发插件提拱了两个方法，  [jQuery](http://caibaojian.com/t/jquery" \o "View all posts in jQuery" \t "http://blog.csdn.net/yummy_go/article/details/_blank).fn.extend();  [jQuery](http://caibaojian.com/t/jquery" \o "View all posts in jQuery" \t "http://blog.csdn.net/yummy_go/article/details/_blank).extend();  jQuery.fn = jQuery.prototype;  $(“#btn1″) 会生成一个 jQuery类的实例,即可以直接调用jQuery.prototype里面的方法。  对于jQuery.extend：  1、jquery.extend(obj):为jQuery类添加类方法，可以理解为添加静态方法  如：  jQuery.extend({  min: function(a, b) { return a < b ? a : b; },  max: function(a, b) { return a > b ? a : b; }  });  jQuery.min(2,3); //  2  jQuery.max(4,5); //  5  jquery.extend( target, object1, [objectN]):用一个或多个其他对象来扩展一个对象，返回被扩展的对象  var settings = { validate: false, limit: 5, name: "foo" };  var options = { validate: true, name: "bar" };  jQuery.extend(settings, options);  结果：settings == { validate: true, limit: 5, name: "bar" }  2、  对于jQuery.fn.extend：对jQuery.prototype进得扩展，就是为jQuery类添加“成员函数”。jQuery类的实例可以使用这个“成员函数”。拓展一个对象到jQuery的 prototype里去，这样的话就是插件机制了  $.fn.extend({      alertWhileClick:function() {            $(this).click(function(){                   alert($(this).val());             });       }  });  $("#input1").alertWhileClick(); | |
| 部件库 widget factory | | | |
| 使用： | | 通过调用 jQuery.widget() 来完成，它带有两个参数：  一个是要创建的插件名称：插件的名称必须包含命名空间，在这个实例中，我们使用了 custom 命名空间。您只能创建一层深的命名空间，因此，custom.progressbar是一个有效的插件名称，而 very.custom.progressbar 不是一个有效的插件名称。  一个是包含支持插件的函数对象。  当插件被调用时，它将创建一个新的插件实例，所有的函数都将在该实例的语境中被执行。这与两种重要方式的标准 jQuery 插件不同。首先，语境是一个对象，不是 DOM 元素。其次，语境总是一个单一的对象，不是一个集合。  $.widget( "custom.progressbar", {  // Default options.  options: {  value: 0  },  \_create: function() {  var progress = this.options.value + "%";  this.element  .addClass( "progressbar" )  .text( progress );  },  // Create a public method.  value: function( value ) {    // No value passed, act as a getter.  if ( value === undefined ) {  return this.options.value;  }    // Value passed, act as a setter.  this.options.value = this.\_constrain( value );  var progress = this.options.value + "%";  this.element.text( progress );  },    // Create a private method.  \_constrain: function( value ) {  if ( value > 100 ) {  value = 100;  }  if ( value < 0 ) {  value = 0;  }  return value;  }  });  部件库（Widget Factory）为我们提供了两个属性：  this.element 是一个包含一个元素的 jQuery 对象。如果我们的插件在包含多个元素的 jQuery 对象上调用，则会为每个元素创建一个单独的插件实例，且每个实例都会有自己的 this.element。  this.options，是一个包含所有插件选项的键名/键值对的哈希（hash）。这些选项可以被传给插件。  当我们调用 jQuery.widget()，它通过给 jQuery.fn（用于创建标准插件的系统） 添加函数来扩展 jQuery。1、函数名称是基于您传给 jQuery.widget() 的名称，不带命名空间 - "progressbar"。   1. 传给插件的选项是在插件实例中获取设置的值。   我们可以为任意一个选项指定默认值。当设计您的 API 时，您应该清楚你的插件的最常见的使用情况，以便您可以设置适当的默认值，且确保使所有的选项真正可选。  调用插件方法  现在我们可以初始化我们的进度条，我们将通过在插件实例上调用方法来执行动作。为了定义一个插件方法，我们只在我们传给jQuery.widget() 的对象中引用函数。我们也可以通过给函数名加下划线前缀来定义 "private" 方法。  为了在插件实例上调用方法，您可以向 jQuery 插件传递方法的名称。如果您调用的方法接受参数，您只需简单地在方法名后面传递这些参数即可。  var bar = $( "<div></div>" )  .appendTo( "body" )  .progressbar({ value: 20 });  // Get the current value.  alert( bar.progressbar( "value" ) );  // Update the value.  bar.progressbar( "value", 50 );  // Get the current value again.  alert( bar.progressbar( "value" ) ); | |
| 使用选项 | | option() 方法是自动提供给插件的。option() 方法允许您在初始化后获取并设置选项。该方法像 jQuery 的 .css() 和 .attr() 方法：您可以只传递一个名称作为取值器来使用，也可以传递一个名称和值作为设置器使用，或者传递一个键名/键值对的哈希来设置多个值。当作为取值器使用时，插件将返回与传入名称相对应的选项的当前值。当作为设置器使用时，插件的 \_setOption 方法将被每个被设置的选项调用。我们可以在我们的插件中指定一个 \_setOption 方法来反应选项更改。对于更改选项要独立执行的动作，我们可以重载 \_setOptions。  $.widget( "custom.progressbar", {  options: {  value: 0  },  \_create: function() {  this.options.value = this.\_constrain(this.options.value);  this.element.addClass( "progressbar" );  this.refresh();  },  \_setOption: function( key, value ) {  if ( key === "value" ) {  value = this.\_constrain( value );  }  this.\_super( key, value );  },  \_setOptions: function( options ) {  this.\_super( options );  this.refresh();  },  refresh: function() {  var progress = this.options.value + "%";  this.element.text( progress );  },  \_constrain: function( value ) {  if ( value > 100 ) {  value = 100;  }  if ( value < 0 ) {  value = 0;  }  return value;  }  }); | |
| 添加回调 | | 最简单的扩展插件的方法是添加回调，这样用户就可以在插件状态发生变化时做出反应。我们可以看下面的实例如何在进度达到 100% 时添加回调到进度条。\_trigger() 方法有三个参数：回调名称，一个启动回调的 jQuery 事件对象，以及一个与事件相关的数据哈希。回调名称是唯一一个必需的参数，但是对于想要在插件上实现自定义功能的用户，其他的参数是非常有用的。例如，如果我们创建一个可拖拽插件，我们可以在触发拖拽回调时传递 mousemove 事件，这将允许用户对基于由事件对象提供的 x/y 坐标上的拖拽做出反应。请注意，传递到\_trigger() 的原始的事件必须是一个 jQuery 事件，而不是一个原生的浏览器事件。  $.widget( "custom.progressbar", {  options: {  value: 0  },  \_create: function() {  this.options.value = this.\_constrain(this.options.value);  this.element.addClass( "progressbar" );  this.refresh();  },  \_setOption: function( key, value ) {  if ( key === "value" ) {  value = this.\_constrain( value );  }  this.\_super( key, value );  },  \_setOptions: function( options ) {  this.\_super( options );  this.refresh();  },  refresh: function() {  var progress = this.options.value + "%";  this.element.text( progress );  if ( this.options.value == 100 ) {  this.\_trigger( "complete", null, { value: 100 } );  }  },  \_constrain: function( value ) {  if ( value > 100 ) {  value = 100;  }  if ( value < 0 ) {  value = 0;  }  return value;  }  });  回调函数本质上只是附加选项，所以您可以像其他选项一样获取并设置它们。无论何时执行回调，都会有一个相对应的事件被触发。事件类型是通过连接插件的名称和回调函数名称确定的。回调和事件都接受两个相同的参数：一个事件对象和一个与事件相关的数据哈希，具体如下面实例所示。您的插件可能需要包含防止用户使用的功能，为了做到这点，最好的方法就是创建爱你可撤销的回调。用户可以撤销回调或者相关的事件，与他们撤销任何一个原生事件一样，都是通过调用 event.preventDefault() 或返回 false 来实现的。如果用户撤销回调，\_trigger() 方法将返回 false，这样您就能在插件内实现合适的功能。  var bar = $( "<div></div>" )  .appendTo( "body" )  .progressbar({  complete: function( event, data ) {  alert( "Callbacks are great!" );  }  })  .bind( "progressbarcomplete", function( event, data ) {  alert( "Events bubble and support many handlers for extreme flexibility." );  alert( "The progress bar value is " + data.value );  });    bar.progressbar( "option", "value", 100 ); | |
| 本质 | | 现在我们已经看到如何使用部件库（Widget Factory）创建一个插件，接下来让我们看看它实际上是如何工作的。  当您调用jQuery.widget() 时，它将为插件创建一个构造函数，并设置您为插件实例传入的作为原型的对象。所有自动添加到插件的功能都来自一个基本的小部件原型，该原型定义为 jQuery.Widget.prototype。  当创建插件实例时，会使用 jQuery.data 把它存储在原始的 DOM 元素上，插件名作为键名。由于插件实例直接链接到 DOM 元素上，您可以直接访问插件实例，而不需要遍历插件方法。这将允许您直接在插件实例上调用方法，而不需要传递字符串形式的方法名，同时您也可以直接访问插件的属性。  var bar = $( "<div></div>" )  .appendTo( "body" )  .progressbar()  .data( "progressbar" );  // Call a method directly on the plugin instance.  bar.option( "value", 50 );  // Access properties on the plugin instance.  alert( bar.options.value );  您也可以在不遍历插件方法的情况下创建一个实例，通过选项和元素直接调用构造函数即可：  var bar = $.custom.progressbar( {}, $( "<div></div>" ).appendTo( "body") );  // Same result as before.  alert( bar.options.value );  插件有构造函数和原型的最大好处是易于扩展插件。通过添加或修改插件原型上的方法，我们可以修改插件所有实例的行为。例如，如果我们想要向进度条添加一个方法来重置进度为 0%，我们可以向原型添加这个方法，它将在所有插件实例上可调用。  $.custom.progressbar.prototype.reset = function() {  this.\_setOption( "value", 0 );  }; | |
| 清理 | | 在某些情况下，允许用户应用插件，然后再取消应用。您可以通过 \_destroy() 方法做到这一点。在 \_destroy() 方法内，您应该撤销在初始化和后期使用期间插件所做的一切动作。\_destroy() 是通过 .destroy() 方法被调用的，.destroy() 方法是在插件实例绑定的元素从 DOM 上移除时被自动调用的，所以这可被用于垃圾回收。基本的 .destroy() 方法也处理一些常用的清理操作，比如从小部件的 DOM 元素上移除实例引用，从元素上解除绑定小部件命名空间中的所有事件，解除绑定所有使用 \_bind() 添加的事件。  $.widget( "custom.progressbar", {  options: {  value: 0  },  \_create: function() {  this.options.value = this.\_constrain(this.options.value);  this.element.addClass( "progressbar" );  this.refresh();  },  \_setOption: function( key, value ) {  if ( key === "value" ) {  value = this.\_constrain( value );  }  this.\_super( key, value );  },  \_setOptions: function( options ) {  this.\_super( options );  this.refresh();  },  refresh: function() {  var progress = this.options.value + "%";  this.element.text( progress );  if ( this.options.value == 100 ) {  this.\_trigger( "complete", null, { value: 100 } );  }  },  \_constrain: function( value ) {  if ( value > 100 ) {  value = 100;  }  if ( value < 0 ) {  value = 0;  }  return value;  },  \_destroy: function() {  this.element  .removeClass( "progressbar" )  .text( "" );  }  });  部件库（Widget Factory）只是创建有状态插件的一种方式。这里还有一些其他不同的模型可以使用，且每个都有各自的优势和劣势。部件库（Widget Factory）解决了很多常见的问题，且大大提高了效率，同时也大大提高了代码的重用性，使它适合于 jQuery UI 及其他有状态的插件。  请注意，在本章节中我们使用了 custom 命名空间。ui 命名空间被官方的 jQuery UI 插件保留。当创建您自己的插件时，您应该创建自己的命名空间。这样才能更清楚插件来自哪里，属于哪个范围。 | |
| jQuery UI 通过部件库（Widget Factory）扩展小部件 | | | |
| 简单的方法 | | 在 "custom" 命名空间中创建一个 "superDialog" 小部件。  $.widget( "custom.superDialog", {} );  为了支持扩展，$.widget() 可选性地接受作为父部件使用的小部件的构造函数。当指定一个父部件时，把它作为第二个参数进行传递，放在小部件名称后面，在小部件原型对象前面。  就像上面的实例，下面也要在 "custom" 命名空间中创建一个 "superDialog" 小部件。但是这次传递的是 [jQuery UI 的 dialog（对话框）小部件](http://www.runoob.com/jqueryui/example-dialog.html) 的构造函数（$.ui.dialog），表示 superDialog 小部件应该使用 jQuery UI 的 dialog（对话框）小部件作为父部件。  $.widget( "custom.superDialog", $.ui.dialog, {} );  在这里，superDialog 和 dialog 两个小部件实质上是等价的，只是名称和命名空间不同而已。为了让我们新的小部件更具特点，我们可以添加一些方法到它的原型对象上。  小部件的原型对象是传递给 $.widget() 的最后一个参数。到目前为止，我们的实例使用的是一个空的对象。现在让我们给这个对象添加一个方法：  $.widget( "custom.superDialog", $.ui.dialog, {  red: function() {  this.element.css( "color", "red" );  }  });  // Create a new <div>, convert it into a superDialog, and call the red() method.  $( "<div>I am red</div>" )  .superDialog()  .superDialog( "red" ); | |
| 扩展已有的方法 | | 有时候，您需要调整或添加已有部件方法的行为。您可以把方法名称指定为原型对象上需要重载的方法名称。下面的实例重载了 dialog（对话框）的 [open() 方法](http://www.runoob.com/jqueryui/api-dialog.html" \l "method-open)。  $.widget( "custom.superDialog", $.ui.dialog, {  open: function() {  console.log( "open" );  }  });  // Create a new <div>, and convert it into a superDialog.  $( "<div>" ).superDialog();  当运行这段代码时，有一个问题。由于我们重载了 open() 的默认行为，所以 dialog（对话框）不再显示在屏幕上。  当我们在原型对象上使用方法，我们实际上是重载了原始的方法，在原型链中使用了一个新的方法。  为了让父部件方法可用，部件库（Widget Factory）提供了两个方法 - \_super() 和 \_superApply()。 | |
| 使用  \_super() 和\_superApply()  来访问父部件 | | [\_super()](http://www.runoob.com/jqueryui/api-jQuery-widget.html" \l "method-_super) 和 [\_superApply()](http://www.runoob.com/jqueryui/api-jQuery-widget.html" \l "method-_superApply) 在父部件中调用了同样的方法。请看下面的实例。就像上一个实例，这个实例也重载了 open() 方法来记录"open"。然而，这次运行 \_super() 是调用了 dialog（对话框）的 open()，并打开对话框。  $.widget( "custom.superDialog", $.ui.dialog, {  open: function() {  console.log( "open" );  // Invoke the parent widget's open().  return this.\_super();  }  });    $( "<div>" ).superDialog();  \_super() 和 \_superApply()类似于Function.prototype.call()和Function.prototype.apply() 方法。  因此，\_super() 接受一个参数列表，\_superApply() 接受一个数组作为参数。  下面的实例演示了这二者之间的不同：  $.widget( "custom.superDialog", $.ui.dialog, {  \_setOption: function( key, value ) {    // Both invoke dialog's setOption() method. \_super() requires the arguments  // be passed as an argument list, \_superApply() as a single array.  this.\_super( key, value );  this.\_superApply( arguments );  }  }); | |
| 小部件（Widgets）和多态性（Polymorphism） | | 当在小部件扩展及它们的插件之间进行交互时候，有一点值得注意，父部件的插件不能用来调用子部件元素上的方法。下面的实例演示了这一点。  $.widget( "custom.superDialog", $.ui.dialog, {} );  var dialog = $( "<div>" ).superDialog();  // This works.  dialog.superDialog( "close" );  // This doesn't.  dialog.dialog( "close" );  上面的实例中，父部件的插件，dialog()，不能调用 superDialog 元素上的 close() 方法。 | |
| 定制个性化实例 | | 目前为止，我们看到的实例都有在小部件原型上扩展的方法。在原型上重载的方法影响了小部件的所有实例。  为了演示这一点，请看下面的实例。dialog（对话框）的两个势力都使用了相同的 open() 方法。  $.widget( "ui.dialog", $.ui.dialog, {  open: function() {  console.log( "open" );  return this.\_super();  }});  // Create two dialogs, both use the same open(), therefore "open" is logged twice.  $( "<div>" ).dialog();  $( "<div>" ).dialog();  有时候，您只需要改变小部件的某个实例的行为。为了做到这点，您需要使用正常的 JavaScript 属性分配，获得对实例的引用，并重载该方法。具体如下面实例所示。  var dialogInstance = $( "<div>" )  .dialog()  **.data( "ui-dialog" );**  // Override the close() method for this dialog  dialogInstance.close = function() {  console.log( "close" );  };  // Create a second dialog  $( "<div>" ).dialog();  // Select both dialogs and call close() on each of them.  // "close" will only be logged once.  **$( ":data(ui-dialog)" ).dialog( "close" );**  个性化实例的重载方法技术是完美的一次性定制。 | |
| API | | | |
| 百叶窗-Blind | | | $( document ).click(function() {  $( "#toggle" ).toggle( "blind" );  });  通过将元素包裹在一个容器内，采用"拉百叶窗"效果来隐藏或显示元素。 |
| 反弹特效-Bounce | | | $( "#toggle" ).toggle( "bounce", { times: 3 ， distance: 100}, "slow" );  [distance][string][20]  [最大的反弹距离，以像素为单位。]  [times][integer][5]  [元素反弹的次数。当隐藏或显示时，会为淡入/淡出效果添加半个反弹] |
| 剪辑特效-Clip | | | $( "#toggle" ).toggle( "clip" );  [direction][String]["vertical "]  [剪辑特效隐藏或显示元素的方向。vertical 剪辑上下边缘，horizontal 剪辑左右边缘] |
| 颜色的动画-animate | | | $( "#toggle" ).click(function() {  $( "#elem" ).animate({  color: "green",  backgroundColor: "rgb( 20, 20, 20 )"  });  }); |
| 消失的方向-drop | | | $( "#toggle" ).toggle( "drop" );  $( "#toggle" ).toggle( "drop",{direction:"down"});  [direction][string]["left"]  [隐藏元素时元素降落的方向，显示元素时元素出现的方向。可能的值：up、down、left、right。] |
| 淡入淡出-fade | | | $( "#toggle" ).toggle( "fade" ); |
| 爆炸-explode | | | $( "#toggle" ).toggle( "explode" );  $( "#toggle" ).toggle( "explode" ,{pieces:99},9000);  [pieces][integer][9]  [爆炸裂开的碎片数目，应该是个平方数，任何其他值被舍入到最近的平方数。] |
| 折叠特效-fold | | | $( "#toggle" ).toggle( "fold" );  [size][nuber string][15]  [被折叠元素的尺寸。]  [horizFirst][Boolean][false]  [当折叠时是否先进行水平方向的折叠。请记得，显示的时候与隐藏的时候顺序相反。] |
| [突出（Highlight](http://www.runoob.com/jqueryui/api-highlight-effect.html) | | | $( "#toggle" ).toggle( "highlight" );  [color][String][#ffff99]  [动画期间使用的背景颜色] |
| 膨胀-puff | | | $( "#toggle" ).toggle( "puff" );  [percent][number][150]  [要缩放到的百分比,小于100则表现为缩小] |
| 缩放-Scale | | | $( "#toggle" ).toggle( "scale" );  [direction][string]["both"]  [特效的方向。可能的值："both"、"vertical" 或 "horizontal"。]  [origin][Array][["middle", "center"]]  [消失点]  [percent][number][]  [要缩放到的百分比。]  [scale][string]["both"]  [元素的哪个区域将被调整尺寸："both"、"box"、"content"。当值为 "box" 时，调整元素的边框（border）和内边距（padding）的尺寸。当值为 "content" 时，调整元素内的所有内容的尺寸。] |
| 震动（Shake | | | $( "#toggle" ).effect( "shake" );  [direction][string]["left"]  ["left" 或 "right" 的值将水平震动元素，"up" 或 "down" 的值将垂直震动元素。该值指定元素沿轴线移动时第一步的方向]  [distance][number][20]  [要震动的距离]  [times][integer][3]  [次数] |
| 跳动特效（Pulsate | | | $( "#toggle" ).toggle( "pulsate" );  [times][integer][3]  [次数] |
| 尺寸特效（Size | | | $( "#toggle" ).effect( "size", {  to: { width: 200, height: 60 }  }, 1000 );  [to]  [object][]  [要调整到的高度和宽度]  [origin]  [arrary][[ "top", "left" ]]  [消失点]  [scale]  [string]["both"]  [元素的哪个区域将被调整尺寸："both"、"box"、"content"。当值为 "box" 时，调整元素的边框（border）和内边距（padding）的尺寸。当值为 "content" 时，调整元素内的所有内容的尺寸。] |
| 滑动特效（Slide | | | $( "#toggle" ).toggle( "slide" );  [direction][string]["both"]  [方向：left"、"right"、"up"、"down"、"both"]  [distance][number][元素的 outerWidth]  [特效的距离。默认为元素的高度（height）还是宽度（width）取决于 direction 参数。可以设置为小于元素的宽度（width）/高度（height）的任意整数] |
|  | | |  |
|  | | |  |
|  | | |  |
|  | | |  |
|  | | |  |
| .hide() | | | .hide( effect [, options ] [, duration ] [, complete ] )  .hide( options ) |
| .show() | | | .show( effect [, options ] [, duration ] [, complete ] )  .show( options ) |
| .toggle() | | |  |
| .removeClass() | | | .removeClass( className [, duration ] [, easing ] [, complete ] )  .removeClass( className [, options ] ) |
| .switchClass() | | | .switchClass( removeClassName, addClassName [, duration ] [, easing ] [, complete ] )  .switchClass( removeClassName, addClassName [, options ] ) |
| .toggleClass() | | | .toggleClass( className [, switch ] [, duration ] [, easing ] [, complete ] )  .toggleClass( className [, switch ] [, options ] ) |
| .effect() | | | $( "div" ).effect( "bounce", "slow" );  $("div").effect( effect [, options ] [, duration ] [, complete ] )  [effect][string][]  [options][object][]  [druation][number/string][]  [complete][Funciton()][]  $("div").effect({effect:"bounce",easing:"swing",duration:400,complete:funciton(){},queue:ture})  $("div").effect( options ); |
| Easings | | | Easing 函数指定动画在不同点上的行进速度。  jQuery 核心带有两个 easings：一个是 linear，整个动画以一个不变的速度行进；另一个是swing  （jQuery 核心默认的 easing），行进速度在动画开始和结束时比在动画中间时稍慢。jQuery UI 提供了一些额外的 easing 函数，范围从摆动行为上的变化到定制特效，比如弹跳。  一些 easings 会在动画中产生负值。根据动画的不同属性，实际值可能为零。例如，您可以把 left 取为负值，但是不能把 height 或 opacity 取为负值。  对各种动画曲线研究：  <http://www.runoob.com/jqueryui/api-easings.html>  linear  swing  easeInQuad  easeOutQuad  easeInOutQuad  easeInCubic  easeOutCubic  easeInOutCubic  easeInQuart  easeOutQuart  easeInOutQuart  easeInQuint  easeOutQuint  easeInOutQuint  easeInExpo  easeOutExpo  easeInOutExpo  easeInSine  easeOutSine  easeInOutSine  easeInCirc  easeOutCirc  easeInOutCirc  easeInElastic  easeOutElastic  easeInOutElastic  easeInBack  easeOutBack  easeInOutBack  easeInBounce  easeOutBounce  easeInOutBounce |
| 转移特效（Transfer | | | $( "div" ).click(function() {  var i = 1 - $( "div" ).index( this );  $( this ).effect( "transfer", { to: $( "div" ).eq( i ) }, 1000 );  });  div.green {  width: 100px;  height: 80px;  background: green;  border: 1px solid black;  position: relative;  }  div.red {  margin-top: 10px;  width: 50px;  height: 30px;  background: red;  border: 1px solid black;  position: relative;  }  .ui-effects-transfer {  background:red;    } |
| 转移特效（Transfer | | | [classname][string][]  [转移的元素将接收的参数化的 class 名称。]  [to][string][]  [jQuery 选择器，要转移到的元素。]  当尝试两个元素之间的可视化交互时非常有用。  转移的元素本身带有 class ui-effects-transfer，其他的样式需要由您来定义，比如添加背景或边框。 |