jQuery学习笔记

邹业盛 2016-01-10 16:55 更新

- jQuery 是如何工作的
    - 1.1. 开始使用jQuery
    - 1.2. jQuery对象与DOM对象之间的转换
- 丰富的选择器
    - 2.1. 常规选择器
    - 2.2. 属性选择器
    - 2.3. 控件选择器
    - 2.4. 其它选择器
- 节点漫游
    - 3.1. 调用链处理
    - 3.2. 子节点
    - 3.3. 兄弟节点
    - 3.4. 父节点
- 元素控制相关
    - 4.1. attributes和properties的区别
    - 4.2. 类与属性控制
    - 4.3. 样式控制
    - 4.4. 结构控制
        - 4.4.1. 文本节点
        - 4.4.2. 子节点
        - 4.4.3. 兄弟节点
        - 4.4.4. 父节点
        - 4.4.5. 复制/删除/替换节点
- 工具函数
    - 5.1. jQuery对象序列
    - 5.2. 通用工具
- 上下文绑定
- 把数据存到节点中
- 事件处理

        - 8.1. 事件绑定
        - 8.2. 事件触发
        - 8.3. 事件类型
        - 8.4. 事件对象
    - AJAX
        - 9.1. 请求与回调
        - 9.2. 请求的状态
        - 9.3. 工具函数
    - 泛化回调
        - 10.1. Deferred
        - 10.2. Callbacks

历史：

- 原文发于2010年11月，先把旧文重新排过来，计划近期会修订此文。
- 此文于2013年1月，参照 jQuery 1.8.3 进行了修订。

1. jQuery 是如何工作的

1.1. 开始使用jQuery

jQuery 本身只有一个 js 文件，所以，要使用它，就和使用其它的 js 文件一样，直接将它引入就可以使用了。

```
<script type="text/javascript" src="jquery-1.8.3.js"></script>
```

但是，要注意一点，因为 jQuery 大部分功能需要根据文档的 DOM 模型来工作，所以，它首先需要正确地解析到整个文档的 DOM 模型结构。为此，一般，我们使用 jQuery 所做的工作都需要在整个文档被浏览器完全加载完毕后才开始进行：

```
<script>
$(document).ready(function(){
  alert('Hello World!');
  $("p").click(function(event){
    alert('Thanks for visiting!');
  });
});
</script>
```

$ 是在 jQuery 中被定义的一个函数，它可以简单方便地取到相关结点。
$(document).ready 是一个事件绑定，这在文档加载完毕后被调用。

事实上 $() 等于 jQuery() 即 jQuery 的核心函数的一个简写。

之后的 javascript 代码，都默认是写在 $(document).ready 这个事件绑定当中的。

当然， $(document).ready() 可以简写成 $() 。

前面说了， $ 是一个在 jQuery 中被使用了的一个变量名，如果因为某些原因，你不能让 jQuery 使用它，那么你可以使用 jQuery.noConflict() 做到这一点，它的返回值就是 jQuery 这个对象。

```
jQuery.noConflict();
$j = jQuery.noConflict();
```

1.2. jQuery对象与DOM对象之间的转换

通常，使用 $() 得到的是一个 jQuery 对象。它封装了很多 DOM 对象的操作，但是，它和 DOM 对象之间是不同的。比如，如果你要使用 obj.innerHTML ，那只有当 obj 是一个 DOM 对象时才能用，相应地，如果是 jQuery 对象你应该使用 obj.html() 。

从 DOM 对象转到 jQuery 对象： $(obj) 。

从 jQuery 对象转到 DOM 对象： obj[0] 。

比较正规地从 jQuery 对象到 DOM 的转换，是使用 jQuery 对象的 get() 方法：

```
<ul>
  <li id="foo">foo</li>
  <li id="bar">bar</li>
</ul>

$('li').get();
//[<li id="foo">, <li id="bar">]

$('li').get(0);
//<li id="foo">

$('li').get(-1);
//<li id="bar">
```

2. 丰富的选择器

完整的列表在 http://api.jquery.com/category/selectors/ 。

2.1. 常规选择器

$("*")
　　选择所有结点
(“#id”)
　　ID选择器，注意其中的一些特殊字符，比如 .
(“.class”)
　　类选择器

(“tag”)
按标签选择
(“ancestor descendant”)
选择子元素
(“parent > child”)
选择直接子元素
:focus
获取焦点元素
:first-child :last-child
选择第一个/最后一个子元素
:first :last
截取第一个/最后一个符合条件的元素
(“prev + next”)
直接兄弟元素
(“prev ~ siblings”)
兄弟元素
:nth-child()
索引选择，索引从 1 开始 :nth-child(odd) :nth-child(even) :nth-child(4n)

2.2. 属性选择器

[name~="value"]
属性中包括某单词
[name="value"]
属性完全等于指定值
[name!="value"]
属性不等于指定值
[name]
包括有指定属性的元素

2.3. 控件选择器

:checked
选择所有被中的元素

:selected
被选择了的元素
:disabled :enabled
选择被禁用/未禁用的元素
:hidden
选择隐藏元素，不仅是 [type="hidden"]，还有 display: none
:visible
可见控件， visibility: hidden 和 opacity: 0 同样被认为是可见。

:input :button :checkbox :file :image :password :radio :reset :submit :text
具体控件，图像控件是 [type="image"]

2.4. 其它选择器

[name="value"][name2="value2"]
多个 AND 条件
(“selector1, selector2, selectorN”)
多个 OR 条件
:not()
否定选择
(':contains("text")')
包含有指定内容的元素
:eq() :lt() :gt() :even :odd
列表索引选择（不支持负数）
(':has(selector)')
符合条件的再次过滤
:header
选择像 h1,h2,h3 这些标题元素
:only-child
仅有一个子元素的元素
:empty
空元素，即无内容也无子元素
:parent
非空元素

3. 节点漫游

完整列表在 http://api.jquery.com/category/traversing/

通过上面的选择器，我们可以得到希望处理的节点。但是通常，我们还希望得到当前节点的一些相对节点，以便下一步处理，比如“所有子节点”，“下一个兄弟节点”之类的东西。

3.1. 调用链处理

.add()
　　向已有的节点序列中添加新的对象
.andSelf()
　　在调用链中，随时加入原始序列
.eq()
　　指定索引选取节点， 支持负数
.filter() .is() .not() .find() .first() .last() .has()
　　序列选择
.end()
　　链点回溯

```
<ul class="first">
  <li class="foo">list item 1</li>
  <li>list item 2</li>
  <li class="bar">list item 3</li>
</ul>
<ul class="second">
  <li class="foo">list item 1</li>
  <li>list item 2</li>
  <li class="bar">list item 3</li>
</ul>

$('ul.first').find('.foo').css('background-color', 'red')
  .end().find('.bar').css('background-color', 'green');
```

3.2. 子节点

.children()
　　所有的子节点，可加入过滤条件， .children(selector)

3.3. 兄弟节点

.siblings() .next() .nextAll() .nextUntil() .prev() .prevAll() .prevUntil() .closest()
　　兄弟节点选择

3.4. 父节点

.parent() .parents() .parentsUntil()
　　父节点选择

4. 元素控制相关

4.1. attributes和properties的区别

attributes 是 XML 结构中的属性节点概念范畴：

```
<body onload="prettyPrint()"></body>
```

上面的代码表示了 body 这个节点，有一个名为 onload 的 attributes 。

properties 则是对于 DOM 对象的，对象属性概念范畴。这一般就没有直观的表现，但是它总是可以被访问到的，比如：

```
$('body').get(0).tagName;
//BODY
```

虽然 attributes 与 properties 是不同的概念范畴，但是它们对于某些特殊的属性是有共同的访问属性名的，比如 id 。

4.2. 类与属性控制

.addClass() .hasClass() .removeClass()

添加一个类，判断是否有指定类，删除类

```
$('body').addClass('test');
$('body').addClass(function(index, current){return current + 'new'});

$('body').removeClass('test');
$('body').removeClass(function(index, current){return current + ' ' + 'other'});
```

.toggleClass()

类的开关式转换，它的使用方法有多种：

```
<img class="test other" />

$('img').toggleClass(); //对所有类的开关
$('img').toggleClass('test'); //对指定类的开关
$('img').toggleClass(isTrue); //根据isTrue判断所有类的开关
$('img').toggleClass('test', isTrue); //根据isTrue判断指定类的开关

//同 $('img').toggleClass('test') 只是类名由函数返回
$('img').toggleClass(function(index, class, isTrue){return 'name'});

//isTrue会作为函数的第三个参数传入
```

```
$('img').toggleClass(function(index, class, isTrue){return 'name'}, isTrue);
```

.attr()

获取或者设置一个属性值，它的使用方法有多种：

```
<img id="greatphoto" src="brush-seller.jpg" alt="brush seller" />

$('#greatphoto').attr('alt'); //获取属性
$('#greatphoto').attr('alt', 'Beijing Brush Seller'); //设置属性

//同时设置多个属性
$('#greatphoto').attr({
  alt: 'Beijing Brush Seller',
  title: 'photo by Kelly Clark'
});

//设置属性为函数返回值, 函数的上下文为当前元素
$('#greatphoto').attr('title', function(i, val) {
  return val + ' – photo by Kelly Clark';
});
```

.prop()

用法同 .attr() ， 只是对象变成了 properties

.removeAttr() .removeProp()

删除属性

.val()

设置或获取元素的表单值，通常用于表单元素。

```
<input type="hidden" value="test" />

$('input').val();
$('input').val('other');
```

.html()

设置或获取元素的节点文件本值。

```
<div><span>测试</span></div>

$('div').html();
$('div').html('<div>测试</div>');
$('div').html(function(index, old){return old + '<span>另外的内容</span>'});
```

4.3. 样式控制

.css()

获取或设置指定的 css 样式：

```
$('body').css('background-color');
$('body').css('background-color', 'red');
$('body').css('background-color', function(index, value){return value + '1'});
$('body').css({color: 'green', 'background-color': 'red'});
```

.width() .height()

获取或设置元素的宽和高

```
$('body').width();
$('body').width(50);
$('body').width(function(index, value){return value += 10});
```

.innerWidth() .innerHeight() .outerHeight() .outerWidth()

元素的其它尺寸值

.scrollLeft() .scrollTop()

获取或设置滚动条的位置

.offset() .position()

获取元素的坐标。 offset 是相对于 document ， position 是相对于父级元素

4.4. 结构控制

4.4.1. 文本节点

.html() .text()

设置和获取节点的文值。设置时 .text() 会转义标签，获取时 .text() 会移除所有标签。

4.4.2. 子节点

.append() .prepend()

```
<h2>Greetings</h2>
<div class="container">
  <div class="inner">Hello</div>
  <div class="inner">Goodbye</div>
</div>

$('.inner').append('<p>Test</p>');

<h2>Greetings</h2>
<div class="container">
  <div class="inner">
    Hello
    <p>Test</p>
  </div>
  <div class="inner">
    Goodbye
    <p>Test</p>
  </div>
</div>
```

参数可以有多种形式：

```
var $newdiv1 = $('<div id="object1"/>'),
    newdiv2 = document.createElement('div'),
    existingdiv1 = document.getElementById('foo');

$('body').append($newdiv1, [newdiv2, existingdiv1]);
```

4.4.3. 兄弟节点

.after() .before()

```
<div class="container">
  <h2>Greetings</h2>
  <div class="inner">Hello</div>
  <div class="inner">Goodbye</div>
</div>

$('.inner').after('<p>Test</p>');

<div class="container">
  <h2>Greetings</h2>
  <div class="inner">Hello</div>
  <p>Test</p>
  <div class="inner">Goodbye</div>
  <p>Test</p>
</div>
```

4.4.4. 父节点

.wrap() .wrapAll() .wrapInner()

```
<div class="container">
  <div class="inner">Hello</div>
  <div class="inner">Goodbye</div>
</div>

$('.inner').wrap('<div class="new" />');

<div class="container">
  <div class="new">
    <div class="inner">Hello</div>
  </div>
  <div class="new">
    <div class="inner">Goodbye</div>
  </div>
</div>


$('.inner').wrapAll('<div class="new" />');

<div class="container">
  <div class="new">
    <div class="inner">Hello</div>
    <div class="inner">Goodbye</div>
  </div>
</div>


$('.inner').wrapInner('<div class="new" />');

<div class="container">
  <div class="inner">
    <div class="new">Hello</div>
  </div>
  <div class="inner">
    <div class="new">Goodbye</div>
  </div>
</div>
```

4.4.5. 复制/删除/替换节点

.clone()

复制节点，可选参数表示是否处理已绑定的事件与数据

.clone(true)

处理当前节点的事件与数据

.clone(true, true)

处理当前节点及所有子节点的事件与数据

.detach()

暂时移除节点，之后可以再次恢复了指定位置。

.remove()

永久移除节点。

.empty()

清除一个节点的所有内部内容。

.unwrap()

移除节点的父节点。

5. 工具函数

5.1. jQuery对象序列

.map()

遍历所有成员

```
$(':checkbox').map(function() {
  return this.id;
}).get().join(',');


$(':checkbox').map(function(index, node) {
  return node.id;
}).get().join(',');
```

.slice()

序列切片, 支持一个或两个参数，支持负数

```
$('li').slice(2).css('background-color', 'red');
$('li').slice(2, 4).css('background-color', 'red');
$('li').slice(-2, -1).css('background-color', 'red');
```

5.2. 通用工具

$.each() $.map()

遍历列表, $.map() 可以作用于对象。

```
$.each([52, 97], function(index, value) {
  alert(index + ': ' + value);
});

$.map( [0,1,2], function(index, n){
  return n + 4;
});
//[4, 5, 6]

$.map( [0,1,2], function(n){
  return n > 0 ? n + 1 : null;
});
//[2, 3]

$.map( [0,1,2], function(n){
  return [ n, n + 1 ];
});
//[0, 1, 1, 2, 2, 3]

var dimensions = { width: 10, height: 15, length: 20 };
$.map( dimensions, function( value, key ) {
  return value * 2;
});
//[20, 30, 40]
```

```
var dimensions = { width: 10, height: 15, length: 20 },
$.map( dimensions, function( value, key ) {
  return key;
});
//["width", "height", "length"]
```

$.extend()

合并对象，第一个参数表示是否进行递归深入

```
var object = $.extend({}, object1, object2);
var object = $.extend(true, {}, object1, object2);
```

$.merge()

合并列表

```
$.merge( [0,1,2], [2,3,4] )
//[0,1,2,2,3,4]
```

$.grep()

过滤列表，第三个参数表示是否为取反

```
$.grep( [0,1,2], function(array,index){ return n > 0; }); //[1,2]
$.grep( [0,1,2], function(array,index){ return n > 0; }, true); //[0]
```

$.inArray()

存在判断

```
$.inArray( value, array [, fromIndex] )
```

$.isArray() $.isEmptyObject() $.isFunction() $.isNumeric() $.isPlainObject()
$.isWindow() $.isXMLDoc()

类型判断

$.noop()

空函数

$.now()

当前时间戳，值为 (new Date).getTime()

$.parseJson() $.parseXML()

把字符串解析为对象

```
var xml = "<rss version='2.0'><channel><title>RSS Title</title></channel></rss>",
xmlDoc = $.parseXML( xml ),
$xml = $( xmlDoc ),
$title = $xml.find( "title" );
```

$.trim()

去头去尾， $.trim(str)

$.type()

判定参数的类型

```
# If the object is undefined or null,
#then "undefined" or "null" is returned accordingly.

* jQuery.type(undefined) === "undefined"
* jQuery.type() === "undefined"
* jQuery.type(window.notDefined) === "undefined"
* jQuery.type(null) === "null"

# If the object has an internal [[Class]] equivalent to
```

```
#one of the browser's built-in objects, the associated name is returned.
#(More details about this technique.)

* jQuery.type(true) === "boolean"
* jQuery.type(3) === "number"
* jQuery.type("test") === "string"
* jQuery.type(function(){}) === "function"
* jQuery.type([]) === "array"
* jQuery.type(new Date()) === "date"
* jQuery.type(/test/) === "regexp"
```

$.unique()

遍历后去重， $.unique(array)

6. 上下文绑定

上下文绑定在函数型的语言当中起着非常重要的作用。不过， jQuery 中对此似乎并没有怎么强调，它的很多函数都默认处理了上下文，文档中也会告诉你，在某个函数当中， this 是什么东西，我觉得这种做法不太好。

jQuery 中有一个函数专门用于处理上下文绑定，虽然我觉得把 context 放在后面的格式非常别扭，不过好过没有吧。

$.proxy()

为函数绑定上下文

它有两种使用方法，第一种就是最普通的，提供函数，和上下文对象，然后返回一个新函数：

```
$.proxy( function, context )
```

第二种，是提供一个上下文对象，及在这个上下文当中的一个成员名，然后把这个上下文绑定到这个成员之后，再返回这个成员。

```
$.proxy( context, name )
```

看下面的例子，来理解上下文：

```
var o = {
  x: '123',
  f: function(){console.log(this.x)},
}
var go = function(f){f()}

o.f() // 123
go(o.f) // undefined
go($.proxy(o.f, o)) //123
$.proxy(o, 'f')() //123
```

当一个函数被传递之后，它就失去原先的上下文了。

7. 把数据存到节点中

jQuery 提供了一种机制，可以把节点作为数据存储的容器。

$.data()
　　往节点中获取/设置数据
$.removeData()
　　删除数据

在内部实现上， jQuery 会在指定节点添加一个内部标识，以此为 key ，把数据存在内部闭包的一个结构当中。事实上， jQuery 的事件绑定机制也使用了这套数据接口。

```
$.data($('#data').get(0), 'test', '123');
$('#data').data('test', '456');
console.log($.data($('#data').get(0), 'test'));
console.log($('#data').data('test'));
```

8. 事件处理

8.1. 事件绑定

在 jQuery1.7 之后，推荐统一使用 on() 来进行事件绑定。

在讲事件绑定之前，需要首先明确一个概念。通常，我们是需要对具体的节点的相关事件作处理，比如一个按钮被点击了之类的。但这里有一个问题，如果我们能获取到具体的节点，当然没什么说的。不过有时，我们要处理的节点却还没有存在，但是我们又要预定义它的事件处理。比如，我们说在一个 UL 中，每一个 LI 被点击时，都要执行一个函数。但是， UL 中的内容本身可能是变化的，开始时它只有两个 LI ，之后又变成了三个 LI ，多出的那一个 LI 也要能响应事件才行。

上述的情况是一个普遍的需求。我们可以利用事件的冒泡机制来实现目的。我们把事件处理绑定在 UL 上，这样，当 LI 被点击时，会冒泡到上级的 UL ，这样在处理事件时就可以判断出具体是哪个节点的事件，以便作下一步处理。

当然，原理是这样，具体操作时， on() 这个方法已经把一切都封装好了。

.on()
　　绑定事件
.off()

移除绑定

.one()

绑定单次事件

on() 的基本使用方式是： .on(event, handler)

最基本的使用：

```
$('#btn').on('click',
  function(eventObj){
    console.log('haha');
  }
);
```

对于 handler ， 它默认的上下文是触发事件的节点（与行为节点无关）：

```
<div id="btn">
  <div>哈哈</div>
  <div id="btn2">哈哈</div>
  <div>哈哈</div>
</div>

$('#btn').on('click',
  function(eventObj){
    console.log(this);
  }
);
```

你看到是 div#btn 。

当然，使用 $.proxy() 你可以随意控制上下文：

```
$('#btn').on('click',
  $.proxy(function(eventObj){
    console.log(this.a);
  }, {a: 123})
);
```

event 参数还支持通过：

- 以 . 分割的子名字，这些名字父子的名字空间效果。
- 以空格分割的多个事件。

名字空间效果：

```
$('#btn').on('click.my',
  function(eventObj){
    console.log('123');
  }
);

var f = function(){
  $('#btn').off('click.my');
}
```

```
$('#btn').on('click.my.other',
  function(eventObj){
    console.log('123');
  }
);

var f = function(){
  $('#btn').off('click.my');
}
```

多个事件：

```
$('#btn').on('click.my click.other',
  function(eventObj){
    console.log('123');
  }
);

var f = function(){
  $('#btn').off('click.my');
}
```

on() 的扩展使用方式为： .on(events [, selector] [, data], handler(eventObject)) 。

其中的 data 是一个数据存储空间，在 eventObj.data 中访问到。

```
$('#btn').on('click', {a: 123},
  function(eventObj){
    console.log(eventObj.data.a);
  }
);
```

而 selector 则是一个过滤规则，这对于在前面讲过的，在父级节点预定义不存在节点的行为很有用。

```
<div id="btn">
  <span>哈哈</span>
</div>

$('#btn').on('click', 'p',
  function(eventObj){
```

```
    console.log('here');
  }
);
var f = function(){
  $('#btn').append('<p>新的</p>');
}
```

on() 的另外一种调用形式： .on(events-map [, selector] [, data]) ：

```
$('#btn').on(
  {
    'click': function(eventObj){console.log('click')},
    'mousemove': function(eventObj){console.log('move')}
  }
);
```

off() 的使用方式与 on() 完全类似：

```
var f = function(eventObj){
  console.log('here');
}
$('#btn').on('click', f);
$('#btn').off('click');
$('#btn').off('click', '.cls');
$('#btn').off('click', f);
```

8.2. 事件触发

事件的触发有两种方式，一是使用预定的“事件函数”，二是使用 trigger() 或 triggerHandler() 。

预定的“事件函数”就是类似于 .click() .focus() 这些。

```
<input id="btn" />

$('#btn').on('click',
  function(eventObj){
    console.log('here');
  }
);

$('#btn').click();
$('#btn').trigger('click');
```

trigger() 还可以直接接受一个 Event 对象，比如一个死循环：

```
$('#btn').on('click',
  function(eventObj){
    console.log('here');
    $(this).trigger(eventObj);
  }
);
```

trigger() 与 triggerHandler() 的不同之处在于前面是触发事件，而后者是执行绑定函数。这在一些有原生行为的事件上就有明显区别了：

```
$('#btn').on('focus',
  function(eventObj){
    console.log('here');
  }
);

//$('#btn').trigger('focus');
$('#btn').triggerHandler('focus');
```

```

```

trigger() 和 triggerHandler() 也用于触发自定义事件：

```
$('#btn').on('my',
  function(eventObj){
    console.log('here');
  }
);

$('#btn').trigger('my');
```

trigger() 和 triggerHandler() 触发事件时，可以带上参数：

```
$('#btn').on('my',
  function(eventObj, obj){
    console.log(obj);
  }
);

$('#btn').trigger('my', {a: 123});
```

8.3. 事件类型

行为事件：

.click()
    单击
dblclick()
    双击
.blur()

失去焦点时

.change()

值变化时

.focus()

获取焦点时

.focusin()

jQuery 扩展的获取焦点

.focusout()

jQuery 扩展的失去焦点

.resize()

调整大小

.scroll()

滚动

.select()

被选择

.submit()

表单被提交

键盘事件：

.keydown()

按下键

.keyup()

放开键

鼠标事件：

.mousedown()

点下鼠标

.mouseup()

松开鼠标

.mouseover()

光标移入

.mouseout()

光标移出

.mousemove()
光标在之上移动
.mouseleave() .mouseenter()
光标移出移入

页面事件：

.ready()
就绪
.unload()
离开当前页时，针对 window 对象。
.error()
发生错误时
.load()
正在载入

8.4. 事件对象

每一个事件的绑定函数，都接受一个事件对象为参数。这个事件对象当中，包括了很多事件的信息。

event.currentTarget , event.target
事件绑定节点 / 事件的触发节点(冒泡行为)
event.delegateTarget
绑定事件的对象，通常就是 event.currentTarget
event.relatedTarget
相关的节点，主要用于一些转换式的事件。比如鼠标移入，表示它从哪个节点来的
event.which
标明哪个按键触发了事件，鼠标和键盘的键标识统一在这个属性中了
event.preventDefault() , event.isDefaultPrevented()
禁止默认行为
event.stopImmediatePropagation() , event.isImmediatePropagationStopped()
不光禁止冒泡，还终止绑定函数链的继续进行。
event.stopPropagation() , event.isPropagationStopped()

禁止冒泡

event.pageX , event.pageY

事件触发时相对于 document 的鼠标位置

event.namespace

事件触发时的名字空间，比如 trigger('click.namespace')

event.data

额外传入的数据

event.result

上一个绑定函数的返回值

event.timeStamp

事件触发时的时间，其值为 (new Date).getTime()

event.type

事件类型

如果一个绑定函数最后返回了 false ， 则默认是 event.preventDefault() 和 event.stopPropagation() 行为。

9. AJAX

jQuery 对 AJAX 的封装参数比较多，而且，大部分在普通 Web 应用中用得很少，如果真是需要控制到那么精细的地步，我觉得首先应该反思一下设计。jQuery 的 AJAX 封装中，已经包括了两种异步请求类型， XHR ， script , 还差一种 iframe 就齐全了。同时， jQuery 的 AJAX 封装中，包括了对 jsonp 形式的支持。

9.1. 请求与回调

jQuery 的 AJAX ， 核心的请求处理函数只有一个，就是 $.ajax() ， 然后就是一个简单的上层函数。

$.ajax() 的基本使用形式是：

```
jQuery.ajax( settings )
```

settings 是一个对象，里面包含了所有的配置项。

这里，只介绍常用的 settings 项：

url
: 请求的地址。

type
: 请求的方法类型， GET , POST 。 默认是 GET 。

data
: 要发送出去的数据。

dataType
: 服务器返回的数据类型，支持： 'xml', 'html', 'script', 'json', 'jsonp', 'text' 。

success
: 请求成功时调用的处理函数。 success(data, textStatus, jqXHR) 。

context
: 回调函数执行时的上下文

cache
: 默认为 true ， 是否为请求单独添加一个随机参数以防止浏览器缓存

error
: 请求错误时的调用函数。 error(jqXHR, textStatus, errorThrown) ， 第二个参数是表示请求状态的字符串： "timeout", "error", "abort", "parsererror" 。第三个参数是当 HTTP 错误发生时，具体的错误描述： "Not Found", "Internal Server Error." 等。

complete
: 请求结束（无论成功或失败）时的一个回调函数。 complete(jqXHR, textStatus) ， 第二个参数是表示请求状态的字符串： "success", "notmodified", "error", "timeout", "abort", "parsererror" 。

jsonp
: 一个参数名，默认是 callback ， 一般用于指明回调函数名。设置成 false 可以让请求没有 callback 参数。

jsonpCallback
: callback 参数值。默认是自动生成的一个随机值。

对于整套应用来说，其资源请求通常有一套约定的规则，使用 $.ajaxSetup() 可以配置参数的默认值，参数就是上面介绍的那些（不完整）。

$.ajax(options)
AJAX 请求的默认配置。

前面提到过， $.ajax() 是一个核心函数，在其之上，有一些现成的封装，常用的是：

$.get(url [, data] [, success(data, textStatus, jqXHR)] [, dataType])
GET 请求
$.post(url [, data] [, success(data, textStatus, jqXHR)] [, dataType])
POST 请求

9.2. 请求的状态

一个 AJAX 请求可以看成是一个触发了一连串事件的总事件。这样，对于全局的所有 AJAX 请求而言，我们可以在任意节点上，绑定到全局任意 AJAX 请求的每一个事件：

```
$("#loading").ajaxStart(function(){
    $(this).show();
 });
```

.ajaxStart()
请求将要发出时
.ajaxSend()
请求将要发出时(在 .ajaxStart() 后)
.ajaxSuccess()
请求成功
.ajaxError()
请求错误
.ajaxComplete()
请求完成
.ajaxStop()
请求结束(在 .ajaxComplete() 后)

上面这几个 ajax 的全局事件，一般只在 document 上处理。

9.3. 工具函数

.serialize()

解析表单参数项，返回字符串。

```
$('form').submit(function() {
  alert($(this).serialize());
  return false;
});
//a=1&b=2
```

.serializeArray()

解析表单参数项，返回一个列表对象。

```
$('form').submit(function() {
  console.log($(this).serializeArray());
  return false;
});

//
[
  {name: 'a', value: '1'},
  {name: 'b', value: '2'}
]
```

10. 泛化回调

前面已经介绍过的 AJAX 是一种典型的异步回调的过程，我们发出请求，然后定义了当请求

返回时要做什么事。显然，同样的机制可以应用于任何的地方，特别是在于 javascript 这种本来大部分时候都处于异步的运行环境中时，回调的方式更是大有用武之地。

好了，其实我要说的就是在除 AJAX 之外的地方，也应该有机制来支持我们使用异步回调。

10.1. Deferred

Deferred 对象是在 jQuery1.5 中引入的回调管理对象。其作用，大概就是把一堆函数按顺序放入一个调用链，然后根据状态，来依次调用这些函数。 AJAX 的所有操作都是使用它来进行封装的。比如我们定义的，当请求正常返回时，会调用 success 定义的函数，失败时，会调用 error 定义的函数。这里的“失败”，“正常”就是状态，而对应的函数，只是调用链中的一个而且。

先来看一个直观的例子：

```
var obj = $.Deferred(function(a){});
obj.done(function(){console.log('1')});
obj.done(function(){console.log('2')});
obj.resolve();
```

这样，我们就可以按顺序看到 1 , 2 这两个输出了。

总的来说， jQuery 的 Deferred 对象有三个状态： done , fail , process 。

- process 只能先于其它两个状态先被激发。
- done 和 fail 互斥，只能激发一个。
- process 可以被重复激发，而 done 和 fail 只能激发一次。

然后， jQuery 提供了一些函数用于添加回调，激发状态等：

deferred.done()
　　添加一个或多个成功回调。
deferred.fail()
　　添加一个或多个失败回调。
deferred.always()
　　添加一个函数，同时应用于成功和失败。
deferred.progress()
　　添加一个函数用于准备回调。
deferred.then()
　　依次接受三个函数，分别用于成功，失败，准备状态。
deferred.reject()
　　激发失败状态。
deferred.resolve()
　　激发成功状态。
deferred.notify()
　　激发准备状态。

如果一个 Deferred 已经被激发，则新添加的对应的函数会被立即执行。

除了上面的这些操作函数之外， jQuery 还提供了一个 jQuery.when() 的回调管理函数，可以用于方便地管理多个事件并发的情况，先看一个 AJAX 的“原始状态”例子：

```
var defer = $.ajax({
  url: '/json.html',
  dataType: 'json'
});

defer.done(function(data){console.log(data)});
```

.done() 做的事和使用 success 定义是一样的。

当我们需要完成，像“请求A和请求B都完成时，执行函数”之类的需求时，使用 $.when() 就可以了：

```
var defer_1 = $.ajax({
  url: '/json.html',
  dataType: 'json'
});

var defer_2 = $.ajax({
  url: '/jsonp.html',
  dataType: 'jsonp'
});

var new_defer = $.when(defer_1, defer_2);
new_defer.done(function(){console.log('haha')});
```

在 $.when() 中的 Deferred ，只要有一个是 fail ，则整体结果为 fail 。

Deferred 的回调函数的执行顺序与它们的添加顺序一致。

这里特别注意一点，就是 done / fail / always 与 then 的返回值的区别。从功能上看，它们都可以添加回调函数，但是，方法的返回值是不同的。 前组的返回值是原来的那个 defer 对象，而 then 返回的是一个新的 defer 对象。

then 返回新的 defer 这种形式，可以用于方便地实现异步函数的链式调用。

比如对于：

```
var defer = $.ajax({
    url: '/json',
    dataType: 'json'
});
```

如果使用 done 方法：

```
defer.done(function(){
  return $.ajax({
    url: '/json',
    dataType: 'json',
    success: function(){
      console.log('inner')
    }
  });
}).done(function(){
  console.log('here');
});
```

等同于是调用了两次 defer.done ，defer.done ， 注册的两次回调函数依次被执行后，我们看到的输出是：

```
here
inner
```

这是两次 defer.done 的结果，第一个回调函数返回了一个新的 defer 没任何作用。

如果换成 then 方法的话：

```
defer.then(function(){
  return $.ajax({
    url: '/json',
    dataType: 'json',
    success: function(){
      console.log('inner')
    }
  });
}).done(function(){
  console.log('here');
});
```

上面的代码相当于：

```
var new_defer = defer.then(...);
new_defer.done(...);
```

它跟两次 defer.done 是不同的。 new_defer 会在 inner 那里的 defer 被触发时再被触发，所以输出结果是：

```
inner
here
```

更一般地来说 then 的行为，就是前面的注册函数的返回值，会作为后面注册函数的参数值：

```
var defer = $.ajax({
  url: '/json',
  dataType: 'json'
});

defer.then(function(res){
  console.log(res);
  return 1;
}).then(function(res){
  console.log(res);
  return 2;
}).then(function(res){
  console.log(res);
});
```

上面代码的输入结果是：

```
ajax response
1
2
```

10.2. Callbacks

讲了 Deferred ，现在再来看 Callbacks ，这才是本源。事实上， Deferred 机制，只是在 Callbacks 机制的上层进行了一层简单封装而且。 Callbacks 对象，才是真正的 jQuery 中定义的原始的回调管理机制。

```
var obj = $.Callbacks();
obj.add(function(){console.log('1')});
obj.add(function(){console.log('2')});
obj.fire();
```

Callbacks 对象的初始化支持一组控制参数：

$.Callbacks(flags)

初始化一个回调管理对象。 flags 是空格分割的多个字符串，以定义此回调对象的行为：

- once 回调链只能被激发一次
- memory 回调链被激发后，新添加的函数被立即执行
- unique 相同的回调函数只能被添加一次
- stopOnFalse 当有回调函数返回 false 时终止调用链的执行

Callbacks 的控制方法：

callbacks.add()

添加一个或一串回调函数

callbacks.fire()

激发回调

callbacks.remove()

从调用链中移除指定的函数

callbacks.empty()

清空调用链

callbacks.disable()

关闭调用链的继续执行，新添加的函数也不会被执行

callbacks.lock()

锁定调用链，但是如果打开了 memory 的 flag ，新添加的函数仍然会执行

callbacks.has()

检查一个函数是否处于回调链之中

callbacks.fired()

检查调用链是否已经被激发

callbacks.locked()

检查调用链是否被锁定

评论

6条评论 进出自由，我的分享 Disqus 隐私政策 1 登录

推荐 1 推文 分享 最新发布

加入讨论...

通过以下方式登录

或注册一个 DISQUS 帐号

姓名

就会胡闹

4年前

嗯需要更新了,从1.8版开始，函数.ajaxStart()、.ajaxSend()、.ajaxSuccess()、.ajaxError()、.ajaxComplete()、.ajaxStop()只能绑定到document节点，不能绑定到指定节点了，别外deferred.done()｜deferred.fail()与deferred.always()的触发顺序要说一下，补写上去0-0

回复

YS.Zou → 就会胡闹
4年前

非常感谢你的提醒。

ajaxStart 这组事件我去确认了一下，功能上其实是任何节点都可以（源码中也没有限制 document），官方的说法也是 `should` 。不过从实践上来说，只绑定到 document 确实是一种好的方式。

deferred 那里，在你的提醒下，我随便把一直想加的 done 和 then 的区别加上去了。

再次感谢。

回复

Yang
6年前

归纳的不错

回复

123
6年前

不知道为什么1.6以上的版本ajaxStart()不可以用了!为什么呢?

回复

YS.Zou → 123
6年前

2.0.3下我这里没有问题。

回复

姚望
7年前

大概看了二十分钟，对get()、slice()、prop()的印象又深了些，然后对$.map()的了解又更进了一步，以前一直是在有$.each()，两者还是区别蛮大的，然后又学到了个ajaxStart()，收获不小。谢谢作者了。

回复

©2010–2016 zouyesheng.com All rights reserved. Powered by GitHub , txt2tags , MathJax