# jQuery的二把利器

(function (window) {

var jQuery = function () {

return new xxx()

}

window.$ = window.jQuery = jQuery

})(window)

## jQuery核心函数

\* 简称: jQuery函数($/jQuery)

\* jQuery库向外直接暴露的就是$/jQuery

\* 引入jQuery库后, 直接使用$即可

\* 当函数用: $(xxx)

\* 当对象用: $.xxx()

//1. jQuery函数: 直接可用

console.log($, typeof $)

console.log(jQuery===$) // true

1. 作为一般函数调用: $(param)

1). 参数为函数 : 当DOM加载完成后，执行此回调函数

2). 参数为选择器字符串: 查找所有匹配的标签, 并将它们封装成jQuery对象

3). 参数为DOM对象: 将dom对象封装成jQuery对象

4). 参数为html标签字符串 (用得少): 创建标签对象并封装成jQuery对象

2. 作为对象使用: $.xxx()

1). $.each() : 隐式遍历数组

2). $.trim() : 去除两端的空格

/\*需求1. 点击按钮: 显示按钮的文本, 显示一个新的输入框\*/

//1.1). 参数为函数 : 当DOM加载完成后，执行此回调函数

$(function () { // 绑定文档加载完成的监听

// 1.2). 参数为选择器字符串: 查找所有匹配的标签, 并将它们封装成jQuery对象

$('#btn').click(function () { // 绑定点击事件监听

// this是什么? 发生事件的dom元素(<button>)

// alert(this.innerHTML)

// 1.3). 参数为DOM对象: 将dom对象封装成jQuery对象

alert($(this).html())

// 1.4). 参数为html标签字符串 (用得少): 创建标签对象并封装成jQuery对象

$('<input type="text" name="msg3"/><br/>').appendTo('div')

})

})

/\*需求2. 遍历输出数组中所有元素值\*/

var arr = [2, 4, 7]

// 1). $.each() : 隐式遍历数组

$.each(arr, function (index, item) {

console.log(index, item)

})

// 2). $.trim() : 去除两端的空格

var str = ' my atguigu '

// console.log('---'+str.trim()+'---')

console.log('---'+$.trim(str)+'---')

## jQuery核心对象

\* 简称: jQuery对象

\* 得到jQuery对象: 执行jQuery函数返回的就是jQuery对象

\* 使用jQuery对象: $obj.xxx()

//2. jQuery对象: 执行jQuery函数得到它

console.log($() instanceof Object) // true

1. jQuery对象是一个包含所有匹配的任意多个dom元素的伪数组对象

2. 基本行为

\* size()/length: 包含的DOM元素个数

\* [index]/get(index): 得到对应位置的DOM元素

\* each(): 遍历包含的所有DOM元素

\* index(): 得到在所在兄弟元素中的下标

//需求1. 统计一共有多少个按钮

var $buttons = $('button')

/\*size()/length: 包含的DOM元素个数\*/

console.log($buttons.size(), $buttons.length)

//需求2. 取出第2个button的文本

/\*[index]/get(index): 得到对应位置的DOM元素\*/

console.log($buttons[1].innerHTML, $buttons.get(1).innerHTML)

//需求3. 输出所有button标签的文本

/\*each(): 遍历包含的所有DOM元素\*/

/\*$buttons.each(function (index, domEle) {

console.log(index, domEle.innerHTML, this)

})\*/

$buttons.each(function () {

console.log(this.innerHTML)

})

//需求4. 输出'测试三'按钮是所有按钮中的第几个

/\*index(): 得到在所在兄弟元素中的下标\*/

console.log($('#btn3').index()) //2

/\*

1. 伪数组

\* Object对象

\* length属性

\* 数值下标属性

\* 没有数组特别的方法: forEach(), push(), pop(), splice()

\*/

console.log($buttons instanceof Array) // false

// 自定义一个伪数组

var weiArr = {}

weiArr.length = 0

weiArr[0] = 'atguigu'

weiArr.length = 1

weiArr[1] = 123

weiArr.length = 2

for (var i = 0; i < weiArr.length; i++) {

var obj = weiArr[i]

console.log(i, obj)

}

console.log(weiArr.forEach, $buttons.forEach) //undefined, undefined

# 使用jQuery核心函数

## 选择器

### 基本选择器

1. 是什么?

- 有特定格式的字符串

2. 作用

- 用来查找特定页面元素

3. 基本选择器

- #id : id选择器

- element : 元素选择器

- .class : 属性选择器

- \* : 任意标签

- selector1,selector2,selectorN : 取多个选择器的并集(组合选择器)

- selector1selector2selectorN : 取多个选择器的交集(相交选择器)

//1. 选择id为div1的元素

// $('#div1').css('background', 'red')

//2. 选择所有的div元素

// $('div').css('background', 'red')

//3. 选择所有class属性为box的元素

//$('.box').css('background', 'red')

//4. 选择所有的div和span元素

// $('div,span').css('background', 'red')

//5. 选择所有class属性为box的div元素

//$('div.box').css('background', 'red')

//$('\*').css('background', 'red')

### 层次选择器

层次选择器: 查找子元素, 后代元素, 兄弟元素的选择器

1. ancestor descendant

在给定的祖先元素下匹配所有的后代元素

2. parent>child

在给定的父元素下匹配所有的子元素

3. prev+next

匹配所有紧接在 prev 元素后的 next 元素

4. prev~siblings

匹配 prev 元素之后的所有 siblings 元素

//1. 选中ul下所有的的span

// $('ul span').css('background', 'yellow')

//2. 选中ul下所有的子元素span

// $('ul>span').css('background', 'yellow')

//3. 选中class为box的下一个li

$('.box+li').css('background', 'yellow')

//4. 选中ul下的class为box的元素后面的所有兄弟元素

// $('ul .box~\*').css('background', 'yellow')

### 过滤选择器

在原有选择器匹配的元素中进一步进行过滤的选择器

\* 基本

\* 内容

\* 可见性

\* 属性

//1. 选择第一个div

// $('div:first').css('background', 'red')

//2. 选择最后一个class为box的元素

//$('.box:last').css('background', 'red')

//3. 选择所有class属性不为box的div

// $('div:not(.box)').css('background', 'red') //没有class属性也可以

//4. 选择第二个和第三个li元素

// $('li:gt(0):lt(2)').css('background', 'red') // 多个过滤选择器不是同时执行, 而是依次

//$('li:lt(3):gt(0)').css('background', 'red')

//5. 选择内容为BBBBB的li

// $('li:contains("BBBBB")').css('background', 'red')

//6. 选择隐藏的li

// console.log($('li:hidden').length, $('li:hidden')[0])

//7. 选择有title属性的li元素

// $('li[title]').css('background', 'red')

//8. 选择所有属性title为hello的li元素

$('li[title="hello"]').css('background', 'red')

# jQuery核心

// $().each(callback)

let obj = {

name: "tom",

age: 20

}

let arr = [3,5,6];

$(obj).each(function(key , value){

console.log(key,value);

});

console.log("-----$().each()-----")

$(arr).each(function(index , domEle){

console.log(index,domEle);

});

// $.each(obj,callBack)

console.log("------------$.each(obj,callBack)-----------")

$.each(obj,function(key,value){

console.log(key,value)

})

$.each(arr,function(index,domEle){

console.log(index,domEle)

})

区别： 在遍历DOM时，通常用$(selector).each(function(index,element))函数； 因为它有个选择器。在遍历数据时，通常用$.each(dataresource,function(index,element))函数；

// $(obj).size() $(obj).length 这两个返回一样，都是返回jQuery对象中元素的个数

console.log("-----------$(obj).size() $(obj).length--------------")

console.log($(arr).size(),$(obj).length)

// $(obj).get(index) $(obj)[index] 返回的是DOM对象 不传参数则返回全部DOM元素的集合（数组）

console.log("-----------$(obj).get(index) $(obj)[index]--------------")

console.log($($("#box>p").get(0)).html(),$($("#box>p")[1]).html())

// $().index([selector|element])

console.log("-----------$().index()--------------")

console.log($("#text1").index(),$("#box > input").index($("#text2")),$("#btn1").index("#box > input"))

// $().data([key],[value]) $().removeData([name|list])

console.log("-----------$().data()--------------")

$("#box").data("name","box");

console.log($("#box").data("name"))

$("#box").data("more",{first:20,second:33})

console.log($("#box").data("more").first)

$("#box").removeData("name")

// $.fn.extend(object) $.extend(object)

console.log("-----------$.fn.extend() $.extend()--------------")

$.fn.extend({

max : function(a,b){

return a > b ? a : b

},

min : function(a,b){

return a < b ? a : b

}

})

$.extend({

toUpper : function(str){

return str.toUpperCase()

}

})

console.log($("#box").max(22,55),$.toUpper("aaaaaa"))

//jQuery.noConflict([extreme])运行这个函数将变量$的控制权让渡给第一个实现它的那个库。这有助于确保jQuery不会与其他库的$对象发生冲突。 在运行这个函数后，就只能使用jQuery变量访问jQuery对象。例如，在要用到$("div p")的地方，就必须换成jQuery("div p")。 '''注意:'''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。

// 参数 extreme 传入 true 来允许彻底将jQuery变量还原

//描述:

// 将$引用的对象映射回原始的对象。

// jQuery 代码:

jQuery.noConflict();

// 使用 jQuery

jQuery("div p").hide();

// 使用其他库的 $()

$("content").style.display = 'none';

// 描述:

// 恢复使用别名$，然后创建并执行一个函数，在这个函数的作用域中仍然将$作为jQuery的别名来使用。在这个函数中，原来的$对象是无效的。这个函数对于大多数不依赖于其他库的插件都十分有效。

// jQuery 代码:

jQuery.noConflict();

(function($) {

$(function() {

// 使用 $ 作为 jQuery 别名的代码

});

})(jQuery);

// 其他用 $ 作为别名的库的代码

// 描述:

// 创建一个新的别名用以在接下来的库中使用jQuery对象。

// jQuery 代码:

var j = jQuery.noConflict();

// 基于 jQuery 的代码

j("div p").hide();

// 基于其他库的 $() 代码

$("content").style.display = 'none';

// 描述:

// 完全将 jQuery 移到一个新的命名空间。

// jQuery 代码:

var dom = {};

dom.query = jQuery.noConflict(true);

// 结果:

// 新 jQuery 的代码

dom.query("div p").hide();

// 另一个库 $() 的代码

$("content").style.display = 'none';

// 另一个版本 jQuery 的代码

jQuery("div > p").hide();

# 属性

// $().attr(name|properties|key,value|fn) $().removeAttr(name)

console.log("---------------$().attr() $().removeAttr()--------------")

console.log($("#img").attr("src"))

$("#img").attr("src","img/new/2.jpg")

$("#img").attr({

src: "img/new/3.jpg",

title: "imgimg"

})

$("#img").attr("src",function(){

return "img/new/4.jpg"

})

$("#img").attr("myAttr","beautiful")

$("#img").removeAttr("myAttr")

/\* $().prop(name|properties|key,value|fn) $().removeProp(name) 使用方法和attr一样 区别：对于某个 DOM 节点对象，properties 是该对象的所有属性，而 attributes 是该对象对应元素(标签)的属性。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()\*/

// $().addClass([class|fn]) $().removeClass([class|fn])

console.log("--------------- $().addClass() $().removeClass()--------------")

$("#ul1 li:first").addClass("red");

$("#ul1 li:gt(0)").addClass("red big");

$("#ul1 li").addClass(function(index,clazz){

console.log(clazz)

return index + "-" + clazz

});

$("#ul1 li:last").removeClass("big")

$("#ul1 li:eq(1)").removeClass("red big")

$("#ul1 li").removeClass(function(index,clazz){

return index + "-" + clazz

})

// $().toggleClass(class|fn[,sw])

console.log("--------------- $().toggleClass()--------------")

$("#ul1 li").toggleClass("big")

let count = 0

$(document.documentElement).click(function(){

$("#ul1 li:first").toggleClass("red",count++ % 3 == 0)

console.log(count)

})

$("#ul1 li").toggleClass(function(){

return "big"

})

//$().html([val|fn]) $().text()方法和html方法用法一样

console.log("--------------- $().html() --------------")

console.log($("#p1").html())

$("#p2").html("jjjjjjjjjjj")

$("#p1").html(function(index,html){

return "此元素的索引值是：" + index + ",原先的内容是：" + html

})

//$().val([val|fn|arr])

console.log("--------------- $().text() --------------")

$(document).click(function(){

console.log($("#text1").val())

})

// $().val([val|fn|arr])

$("#text1").val("abcdefg")

$("#text1").val(function(index,value){

return "元素的索引是：" + index + ",原先的内容是：" + value

})

// 用于设置选中的项

$("#single").val("Single2");

$("#multiple").val(["Multiple2", "Multiple3"]);

$("input").val(["check2", "radio1"]);

# Css

// $().css(name|pro|[,val|fn])

console.log("------------$().css()----------------")

console.log($("#p1").css("color"))

$("#p1").css("color","green");

$("#p2").css({

color:"red",

fontSize:"25px"

})

$("#p1").css("fontSize",function(index,value){

return parseInt(value) + 10 + "px"

})

// $().offset([coordinates]) 获取匹配元素在当前视口（相对于文档document）的相对偏移。返回的对象包含两个整型属性：top 和 left，以像素计。此方法只对可见元素有效。

console.log("------------$().offset()----------------")

console.log($("#p1").offset().top)

console.log($("#p1").offset().left)

$("#ul1").offset({

left: 100,

top: 800

})

// $().position() 获取匹配元素相对父元素的偏移。返回的对象包含两个整型属性：top 和 left。为精确计算结果，请在补白、边框和填充属性上使用像素单位。此方法只对可见元素有效。

/\*

使用position()方法时事实上是把该元素当绝对定位来处理，获取的是该元素相当于最近的一个拥有绝对或者相对定位的父元素的偏移位置。

使用position()方法时如果其所有的父元素都为默认定位（static）方式，则其处理方式和offset()一样，是当前窗口的相对偏移

使用offset()方法不管该元素如何定位，也不管其父元素如何定位，都是获取的该元素相对于当前视口的偏移

\*/

console.log("------------$().position()----------------")

console.log($("#p1").position().left)

console.log($("#p1").position().top)

// $().scrollTop([val]) $().scrollLeft([val])

console.log("------------$().scrollTop() $().scrollLeft()----------------")

console.log($(document.body).scrollTop())

console.log($(document.documentElement).scrollTop())

$(document.documentElement).scrollTop(100)

// $().width([val|fn]) $().height([val|fn])

console.log("------------$().width() $().height()----------------")

console.log($(document).width())

$("#ul1").height(200)

$("#ul1").height(function(index,height){

return (index+1) \* height

})

// $().innerWidth() $().innerHeight() = $().width() + padding

// $().outerWidth() $().outerHeight() = $().innerWidth() + border

// $().outerWidth(true) $().outerHeight(true) = $().outerWidth() + margin

# 文档处理

// $().append(content|fn) 向每个匹配的元素内部追加内容。这个操作与对指定的元素执行appendChild方法，将它们添加到文档中的情况类似。

// $().prepend(content|fn)向每个匹配的元素内部前置内容。这是向所有匹配元素内部的开始处插入内容的最佳方式。

$("#box").append("<p>在后面插入</p>")

$("#box").prepend("<p>在前面插入</p>")

$("#box").append(function(index,html){

// return index + html

})

$("#box").prepend(function(index,html){

// return index + html

})

// $().appendTo(content|fn) 把所有匹配的元素追加到另一个指定的元素集合中。实际上，使用这个方法是颠倒了常规的$(A).append(B)的操作，即不是把B追加到A中，而是把A追加到B中。在jQuery 1.3.2中，appendTo, prependTo, insertBefore, insertAfter, 和 replaceAll这个几个方法成为一个破坏性操作，返回值是所有被追加的内容，而不仅仅是先前所选中的元素。所以，要选择先前选中的元素，需要使用end()方法

$("#text1").appendTo($("#ul1")); // 等价于 $("#ul1").append ("#text1")

$("<p>ddddddddd</p>").appendTo("#box").end().css("fontSize",25)

//$().prependTo(content|fn) 把所有匹配的元素前置到另一个、指定的元素集合中。实际上，使用这个方法是颠倒了常规的$(A).prepend(B)的操作，即不是把B前置到A中，而是把A前置到B中。在jQuery 1.3.2中，appendTo, prependTo, insertBefore, insertAfter, 和 replaceAll这个几个方法成为一个破坏性操作，要选择先前选中的元素，需要使用end()方法，参见 appendTo 方法的例二。

// $().after(content|fn) 在每个匹配的元素之后插入内容。 $().before() 在每个匹配的元素之前插入内容。

$("#text2").after("<b>ffff</b>")

$("#text3").after(function(){

return "<em>ggggggggggggggg</em>"

})

// $().insertAfter(content) 颠倒了$().after() $().insertBefore(content) 颠倒了$().before()

// $().wrap(html|element|fn)

$("#ul1 li").wrap("<b/>")

$("#ul1").wrap($("<div/>"))

$("#box").wrap(function(){

return "<div class=" + $(this).css("backgroundColor") + "/>" // this 指向 #box 这个标签

})

// $().unwrap() 这个方法将移出元素的父元素。这能快速取消 .wrap()方法的效果。匹配的元素（以及他们的同辈元素）会在DOM结构上替换他们的父元素。

$("#ul1 li").unwrap()

//wrapAll(html|ele) 将所有匹配的元素用单个元素包裹起来.这于 '.wrap()'是不同的，'.wrap()'为每一个匹配的元素都包裹一次。这种包装对于在文档中插入额外的结构化标记最有用，而且它不会破坏原始文档的语义品质。这个函数的原理是检查提供的第一个元素并在它的代码结构中找到最上层的祖先元素－－这个祖先元素就是包装元素。

$("#ul1 li").wrapAll("<p/>")

// $().wrapInner(htm|element|fnl) 将每一个匹配的元素的子内容(包括文本节点)用一个HTML结构包裹起来.这个函数的原理是检查提供的第一个元素（它是由所提供的HTML标记代码动态生成的），并在它的代码结构中找到最上层的祖先元素－－这个祖先元素就是包装元素。

$("#ul1 li").wrapInner(function(){

return "<div class=" + $(this).text() + "/>"

})

// $().replaceWith(content|fn) 将所有匹配的元素替换成指定的HTML或DOM元素。

$("#text1").replaceWith("<div id='text1'>hhhhhhhhhhhhhhhhhh</div>")

// $().replaceAll(selector) 用匹配的元素替换掉所有 selector匹配到的元素。

$("<div id='text2'>kkkkkkkkkkkkkkkkkkkkkkkk</div>").replaceAll("#text2")

// $().empty() 删除匹配的元素集合中所有的子节点。

$("#text1").empty()

// $().remove([expr]) expr 用于筛选元素的jQuery表达式 从DOM中删除所有匹配的元素。这个方法不会把匹配的元素从jQuery对象中删除，因而可以在将来再使用这些匹配的元素。但除了这个元素本身得以保留之外，其他的比如绑定的事件，附加的数据等都会被移除。

$("#ul1 li:first").remove()

$("#box div").remove("#text2")

// $().detach([expr]) expr 用于筛选元素的jQuery表达式 从DOM中删除所有匹配的元素。这个方法不会把匹配的元素从jQuery对象中删除，因而可以在将来再使用这些匹配的元素。与remove()不同的是，所有绑定的事件、附加的数据等都会保留下来。

// $().clone([Even[,deepEven]]) 克隆匹配的DOM元素并且选中这些克隆的副本。

// 参数 even 一个布尔值（true 或者 false）指示事件处理函数是否会被复制。V1.5以上版本默认值是：false

// Even[,deepEven] 1:一个布尔值（true 或者 false）指示事件处理函数是否会被复制。2:一个布尔值，指示是否对事件处理程序和克隆的元素的所有子元素的数据应该被复制。

$("#text3").clone(true,true).appendTo("#ul1")

# 筛选

// $().eq(index|-index) 获取当前链式操作中第N个jQuery对象，返回jQuery对象，当参数大于等于0时为正向选取，比如0代表第一个，1代表第二个。当参数为负数时为反向选取，比如-1为倒数第一个，具体可以看以下示例。类似的有get(index),不过get(index)返回的是DOM对象。

// $().first() 获取第一个元素 $().last() 获取最后个元素 返回的是jQuery对象

console.log("------------------$().first()------------------------")

console.log($.type($("#ul1 li").first()))

$("#ul1 li").first().css("fontSize",20)

// $()hasClass(class) 检查当前的元素是否含有某个特定的类，如果有，则返回true。这其实就是 is("." + class)。

console.log("------------------$()hasClass(class) ------------------------")

console.log($("#box").hasClass("box"))

// $().filter(expr|obj|ele|fn) 筛选出与指定表达式匹配的元素集合。这个方法用于缩小匹配的范围。用逗号分隔多个表达式

console.log("------------------$().filter(expr|obj|ele|fn)------------------------")

console.log( $("#box p").filter("#text2") )

console.log( $("#box p").filter(function(index){

return index == 0

}) )

// $().is(expr|obj|ele|fn) 根据选择器、DOM元素或 jQuery 对象来检测匹配元素集合，如果其中至少有一个元素符合这个给定的表达式就返回true。如果没有元素符合，或者表达式无效，都返回'false'。 '''注意：'''在jQuery 1.3中才对所有表达式提供了支持。在先前版本中，如果提供了复杂的表达式，比如层级选择器（比如 + , ~ 和 > ），始终会返回true

console.log("------------------$().is(expr|obj|ele|fn)------------------------")

console.log($("#box p").eq(1).is("#text2"))

console.log($("#box p").eq(0).is(function(index){

return index === 0

}))

// $().map(callback) 将一组元素转换成其他数组（不论是否是元素数组）

console.log("------------------$().map(callback)------------------------")

console.log( $("#box p").map(function(){

return $(this).text()

}).get() )

// $().has(expr|ele) 保留包含特定后代的元素，去掉那些不含有指定后代的元素。.has()方法将会从给定的jQuery对象中重新创建一组匹配的对象。提供的选择器会一一测试原先那些对象的后代，含有匹配后代的对象将得以保留。

$("#ul1 li").has("ul").css("fontSize")

// $().not(expr|ele|fn) 删除与指定表达式匹配的元素

// slice(start, [end]) start可以是负数，表示倒数

// 选中最后一个元素

$("#box p").slice(-1).css("fontSize",20)

// $().children([expr]) 取得一个包含匹配的元素集合中每一个元素的所有子元素的元素集合。可以通过可选的表达式来过滤所匹配的子元素。注意：parents()将查找所有祖辈元素，而children()只考虑子元素而不考虑所有后代元素。

// $().closest(expr|object|element) jQuery 1.3新增。从元素本身开始，逐级向上级元素匹配，并返回最先匹配的元素。closest会首先检查当前元素是否匹配，如果匹配则直接返回元素本身。如果不匹配则向上查找父元素，一层一层往上，直到找到匹配选择器的元素。如果什么都没找到则返回一个空的jQuery对象。closest和parents的主要区别是：1，前者从当前元素开始匹配寻找，后者从父元素开始匹配寻找；2，前者逐级向上查找，直到发现匹配的元素后就停止了，后者一直向上查找直到根元素，然后把这些元素放进一个临时集合中，再用给定的选择器表达式去过滤；3，前者返回0或1个元素，后者可能包含0个，1个，或者多个元素。

//$().find(expr|obj|ele) 搜索所有与指定表达式匹配的元素。这个函数是找出正在处理的元素的后代元素的好方法。所有搜索都依靠jQuery表达式来完成。这个表达式可以使用CSS1-3的选择器语法来写。

// $().next([expr]) 取得一个包含匹配的元素集合中每一个元素紧邻的后面同辈元素的元素集合。这个函数只返回后面那个紧邻的同辈元素，而不是后面所有的同辈元素（可以使用nextAll）。可以用一个可选的表达式进行筛选。

// $().nextAll([expr])查找当前元素之后所有的同辈元素。 可以用表达式过滤

// $().nextUntil([exp|ele][,fil]) 查找当前元素之后所有的同辈元素，直到遇到匹配的那个元素为止。如果提供的jQuery代表了一组DOM元素，.nextUntil()方法也能让我们找遍所有元素所在的DOM树，直到遇到了一个跟提供的参数匹配的元素的时候才会停下来。这个新jQuery对象里包含了下面所有找到的同辈元素，但不包括那个选择器匹配到的元素。如果没有选择器匹配到，或者没有提供参数，那么跟在后面的所有同辈元素都会被选中。这就跟用没有提供参数的 .nextAll() 效果一样。

// 参数 ： expr: 用于筛选祖先元素的表达式。filter: 一个字符串，其中包含一个选择表达式匹配元素(表示将要被匹配到的元素的表达式)。

//element: 用于筛选祖先元素的DOM元素。filter: 一个字符串，其中包含一个选择表达式匹配元素(表示将要被匹配到的元素的表达式)

// $().offsetParent() 返回第一个匹配元素用于定位的父节点。这返回父元素中第一个其position设为relative或者absolute的元素。此方法仅对可见元素有效。

// $().parent([expr]) 取得一个包含着所有匹配元素的唯一父元素的元素集合。你可以使用可选的表达式来筛选。

// $().parents([expr]) 取得一个包含着所有匹配元素的祖先元素的元素集合（不包含根元素）。可以通过一个可选的表达式进行筛选。

// $().parentsUntil([expr|element][,filter]) 查找当前元素的所有的父辈元素，直到遇到匹配的那个元素为止。如果提供的jQuery代表了一组DOM元素，.parentsUntil()方法也能让我们找遍所有元素的祖先元素，直到遇到了一个跟提供的参数匹配的元素的时候才会停下来。这个返回的jQuery对象里包含了下面所有找到的父辈元素，但不包括那个选择器匹配到的元素。

// 参数：expr: 用于筛选祖先元素的表达式 filter: 一个字符串，其中包含一个选择表达式匹配元素。

// element:用于筛选祖先元素的DOM元素 filter: 一个字符串，其中包含一个选择表达式匹配元素。

//$().prev([expr]) 取得一个包含匹配的元素集合中每一个元素紧邻的前一个同辈元素的元素集合。可以用一个可选的表达式进行筛选。只有紧邻的同辈元素会被匹配到，而不是前面所有的同辈元素。

// $().prevAll([expr]) 查找当前元素之前所有的同辈元素 可以用表达式过滤。

// $().prevUntil([exp|ele][,fil]) 查找当前元素之前所有的同辈元素，直到遇到匹配的那个元素为止。如果提供的jQuery代表了一组DOM元素，.prevUntil()方法也能让我们找遍所有元素所在的DOM树，直到遇到了一个跟提供的参数匹配的元素的时候才会停下来。这个新jQuery对象里包含了前面所有找到的同辈元素，但不包括那个选择器匹配到的元素。如果没有选择器匹配到，或者没有提供参数，那么排在前面的所有同辈元素都会被选中。这就跟用没有提供参数的 .prevAll()效果一样。

// 参数：expr: 用于筛选祖先元素的表达式 filter: 一个字符串，其中包含一个选择表达式匹配元素。

// element:用于筛选祖先元素的DOM元素 filter: 一个字符串，其中包含一个选择表达式匹配元素。

// $().siblings([expr]) 取得一个包含匹配的元素集合中每一个元素的所有唯一同辈元素的元素集合。可以用可选的表达式进行筛选。

//add(expr|ele|html|obj[,con]) 把与表达式匹配的元素添加到jQuery对象中。这个函数可以用于连接分别与两个表达式匹配的元素结果集。jQuery 1.4 中， .add()方法返回的结果将始终以元素在HTML文档中出现的顺序来排序，而不再是简单的添加。jQuery 1.9 中， .add()方法返回的结果总是按照节点在document（文档）中的顺序排列。在1.9之前，如果上下文或输入的集合中任何一个以脱离文档的节点（未在文档中）开始，使用.add()方法节点不会按照document（文档）中的顺序排序。现在，返回的节点按照文档中的顺序排序，并且脱离文档的节点被放置在集合的末尾。

$("#box p").add("#ul1 li:first").css("color","green")

// $().addBack() 添加堆栈中元素集合到当前集合，一个选择性的过滤选择器。如上所述在讨论中的.end()， jQuery对象维护一个堆栈内部来跟踪匹配的元素集合的变化。当一个DOM遍历方法被调用时，新的元素集合推入到堆栈中。 如果还需要包含先前的元素集合，.addBack() 可以提供帮助。

$("#ul1 ul li:eq(1)").nextAll().addBack().css("color","red")

// contents() 查找匹配元素内部所有的子节点（包括文本节点）。如果元素是一个iframe，则查找文档内容

$("#box p").contents().wrap("<b/>")

// $().end() 回到最近的一个"破坏性"操作之前。即，将匹配的元素列表变为前一次的状态。如果之前没有破坏性操作，则返回一个空集。所谓的"破坏性"就是指任何改变所匹配的jQuery元素的操作。这包括在 Traversing 中任何返回一个jQuery对象的函数--'add', 'andSelf', 'children', 'filter', 'find', 'map', 'next', 'nextAll', 'not', 'parent', 'parents', 'prev', 'prevAll', 'siblings' and 'slice'--再加上 Manipulation 中的 'clone'。

$("#box").find("#text1").end()

# 事件

// $().ready(fn) 当DOM载入就绪可以查询及操纵时绑定一个要执行的函数。这是事件模块中最重要的一个函数，因为它可以极大地提高web应用程序的响应速度。简单地说，这个方法纯粹是对向window.load事件注册事件的替代方法。通过使用这个方法，可以在DOM载入就绪能够读取并操纵时立即调用你所绑定的函数，而99.99%的JavaScript函数都需要在那一刻执行。有一个参数－－对jQuery函数的引用－－会传递到这个ready事件处理函数中。可以给这个参数任意起一个名字，并因此可以不再担心命名冲突而放心地使用$别名。请确保在 <body> 元素的onload事件中没有注册函数，否则不会触发+$(document).ready()事件。可以在同一个页面中无限次地使用$(document).ready()事件。其中注册的函数会按照（代码中的）先后顺序依次执行。

// 在DOM加载完成时运行的代码，可以这样写：

$(document).ready(function(){

})

// 简写：

$(function($){

})

// $().on(events,[selector],[data],fn) 在选择元素上绑定一个或多个事件的事件处理函数。on()方法绑定事件处理程序到当前选定的jQuery对象中的元素。在jQuery 1.7中，.on()方法 提供绑定事件处理程序所需的所有功能。帮助从旧的jQuery事件方法转换，see .bind(), .delegate(), 和 .live(). 要删除的.on()绑定的事件，请参阅.off()。要附加一个事件，只运行一次，然后删除自己， 请参阅.one()

/\*

events:一个或多个用空格分隔的事件类型和可选的命名空间，如"click"或"keydown.myPlugin" 。

selector:一个选择器字符串用于过滤器的触发事件的选择器元素的后代。如果选择的< null或省略，当它到达选定的元素，事件总是触发。

data:当一个事件被触发时要传递event.data给事件处理函数。

fn:该事件被触发时执行的函数。 false 值也可以做一个函数的简写，返回false。

\*/

$("#btn1").on("click mouseover",function(){

console.log("click or mouseover")

return false

})

// 将第四个参数设置为false会使默认的动作失效。

// $("#btn1").on("click",false)

$("#btn2").on("click",{first:1,second:2},function(e){

console.log(e.data.first,e.data.second)

})

// 绑定多个事件处理函数

$("#btn3").on({

click:function(){

alert("click")

},

mouseout:function(){

alert("mouseout")

}

})

//事件绑定

let index = 0;

let lis = $("#ul1 li")

$("#ul1").on("click","li",{msg:"abcdefg"},function(e){

$(lis[index]).css("backgroundColor","")

$(this).css("backgroundColor","red")

index = $(this).index()

})

// $().off(events,[selector],[fn]) 在选择元素上移除一个或多个事件的事件处理函数。off() 方法移除用.on()绑定的事件处理程序

//移除所有绑定的事件函数

$("#btn1").off()

//移除所有的事件委托函数

$("#ul1").off("click","\*\*")

//移除指定的事件处理函数

function fn(){

alert("fn")

}

$("#btn1").on("click",fn)

$("#btn1").off("click",fn)

// $().bind() 如果使用on的时候，不设置selector，那么on与bind就没有区别了。 on绑定的事件处理函数，对于未来新增的元素一样可以，和delegate效果相同，而bind则不行。

// $().one(type,[data],fn) 为每一个匹配元素的特定事件（像click）绑定一个一次性的事件处理函数。为每一个匹配元素的特定事件（像click）绑定一个一次性的事件处理函数。在每个对象上，这个事件处理函数只会被执行一次。其他规则与bind()函数相同。这个事件处理函数会接收到一个事件对象，可以通过它来阻止（浏览器）默认的行为。如果既想取消默认的行为，又想阻止事件起泡，这个事件处理函数必须返回false。多数情况下，可以把事件处理函数定义为匿名函数（见示例一）。在不可能定义匿名函数的情况下，可以传递一个可选的数据对象作为第二个参数（而事件处理函数则作为第三个参数）

// $().trigger(type,[data]) 在每一个匹配的元素上触发某类事件。这个函数也会导致浏览器同名的默认行为的执行。比如，如果用trigger()触发一个'submit'，则同样会导致浏览器提交表单。如果要阻止这种默认行为，应返回false。

/\*

type:一个事件对象或者要触发的事件类型

data:传递给事件处理函数的附加参数

\*/

//给一个事件传递参数

$("#btn1").on("click",function(e,a,b){

console.log(a,b)

}).trigger("click",["hello","world"])

$("#btn1").on("myEvent",function(){

console.log("myEvent")

})

$("#btn1").trigger("myEvent")

// triggerHandler(type, [data])

/\*

这个特别的方法将会触发指定的事件类型上所有绑定的处理函数。但不会执行浏览器默认动作，也不会产生事件冒泡。

这个方法的行为表现与trigger类似，但有以下三个主要区别：

\* 第一，他不会触发浏览器默认事件。

\* 第二，只触发jQuery对象集合中第一个元素的事件处理函数。

\* 第三，这个方法的返回的是事件处理函数的返回值，而不是据有可链性的jQuery对象。此外，如果最开始的jQuery对象集合为空，则这个方法返回 undefined 。

\*/

$("#btn1").triggerHandler("click",["trigger","Handler"])

// $().unbind(type,[data|fn]])

/\*

bind()的反向操作，从每一个匹配的元素中删除绑定的事件。

如果没有参数，则删除所有绑定的事件。

你可以将你用bind()注册的自定义事件取消绑定。

如果提供了事件类型作为参数，则只删除该类型的绑定事件。

如果把在绑定时传递的处理函数作为第二个参数，则只有这个特定的事件处理函数会被删除。

\*/

// $().delegate(selector,[type],[data],fn) 为指定的元素（属于被选元素的子元素）添加一个或多个事件处理程序，并规定当这些事件发生时运行的函数。使用 delegate() 方法的事件处理程序适用于当前或未来的元素（比如由脚本创建的新元素）。（也是事件代理）

$("#ul1").delegate("li","mouseover",function(){

$(this).css("color","red")

})

// $().undelegate([selector,[type],fn]) 删除由 delegate() 方法添加的一个或多个事件处理程序。

// $().hover([over,]out) 一个模仿悬停事件（鼠标移动到一个对象上面及移出这个对象）的方法。这是一个自定义的方法，它为频繁使用的任务提供了一种“保持在其中”的状态。当鼠标移动到一个匹配的元素上面时，会触发指定的第一个函数。当鼠标移出这个元素时，会触发指定的第二个函数。而且，会伴随着对鼠标是否仍然处在特定元素中的检测（例如，处在div中的图像），如果是，则会继续保持“悬停”状态，而不触发移出事件（修正了使用mouseout事件的一个常见错误）。

$("#ul1 li").hover(function(){

$(this).css("fontSize",20)

},function(){

$(this).css("fontSize",16)

})

// $().toggle([speed],[easing],[fn]) 用于绑定两个或多个事件处理器函数，以响应被选元素的轮流的 click 事件。如果元素是可见的，切换为隐藏的；如果元素是隐藏的，切换为可见的。

/\*

[speed] [,fn]

speed: 隐藏/显示 效果的速度。默认是 "0"毫秒。可能的值：slow，normal，fast。"

fn:在动画完成时执行的函数，每个元素执行一次。

[speed], [easing ], [fn ]

speed: 隐藏/显示 效果的速度。默认是 "0"毫秒。可能的值：slow，normal，fast。"

easing:(Optional) 用来指定切换效果，默认是"swing"，可用参数"linear"

fn:在动画完成时执行的函数，每个元素执行一次。

\*/

$("#btn4").click(function(){

$("#div1").toggle("slow","linear",function(){

console.log("div切换成功")

})

})

// $().change([[data],fn]) 当元素的值发生改变时，会发生 change 事件。该事件仅适用于文本域（text field），以及 textarea 和 select 元素。当用于 select 元素时，change 事件会在选择某个选项时发生。当用于 text field 或 text area 时，该事件会在元素失去焦点时发生。

/\*

$().blur([[data],fn])

$().click([[data],fn])

$().dbclick([[data],fn])

$().error([[data],fn])如果异常是由window对象抛出，事件处理函数将会被传入三个参数：

1. 描述事件的信息 ("varName is not defined", "missing operator in expression", 等等.),

2. 包含错误的文档的完整URL

3. 异常发生的行数 如果事件处理函数返回true，则表示事件已经被处理，浏览器将认为没有异常。

$(window).error(function(msg, url, line){

jQuery.post("js\_error\_log.php", { msg: msg, url: url, line: line });

});

\*/

// $().focusin() 当元素获得焦点时，触发 focusin 事件。focusin事件跟focus事件区别在于，他可以在父元素上检测子元素获取焦点的情况。

$("#p1").focus(function(){

console.log("fufuffu") // 点击文本框无响应

})

$("#p1").focusin(function(){

console.log("input1") // 点击文本框有响应

})

// $().focusout() 当元素失去焦点时触发 focusout 事件。focusout事件跟blur事件区别在于，他可以在父元素上检测子元素失去焦点的情况。

$("#p1").blur(function(){

console.log("fu失去焦点") // 文本框失去焦点无响应

})

$("#p1").focusout(function(){

console.log("sub失去焦点") // 文本框失去焦点有响应

})

/\*

$().keydown([[data],fn])

$().keypress([[data],fn])

$().keyup([[data],fn])

$().mousedown([[data],fn])

$().mouseenter([[data],fn]) 当鼠标指针穿过元素时，会发生 mouseenter 事件。该事件大多数时候会与mouseleave 事件一起使用。与 mouseover 事件不同，只有在鼠标指针穿过被选元素时，才会触发 mouseenter 事件。如果鼠标指针穿过任何子元素，同样会触发 mouseover 事件

$().mouseleave([[data],fn]) 当鼠标指针离开元素时，会发生 mouseleave 事件。该事件大多数时候会与mouseenter 事件一起使用。与 mouseout 事件不同，只有在鼠标指针离开被选元素时，才会触发 mouseleave 事件。如果鼠标指针离开任何子元素，同样会触发 mouseout 事件。

$().mousemove([[data],fn])

$().mouseover([[data],fn])

$().mouseout([[data],fn])

$().mouseup([[data],fn])

$().resize([[data],fn]) 当调整浏览器窗口的大小时，发生 resize 事件。

$().scroll([[data],fn]) 当用户滚动指定的元素时，会发生 scroll 事件。scroll 事件适用于所有可滚动的元素和 window 对象（浏览器窗口）。

$().select([[data],fn])当 textarea 或文本类型的 input 元素中的文本被选择时，会发生 select 事件。这个函数会调用执行绑定到select事件的所有函数，包括浏览器的默认行为。可以通过在某个绑定的函数中返回false来防止触发浏览器的默认行为。

$().submit([[data],fn])当提交表单时，会发生 submit 事件。该事件只适用于表单元素。

$().unload([[data],fn])

在当用户离开页面时，会发生 unload 事件。

具体来说，当发生以下情况时，会发出 unload 事件：

•点击某个离开页面的链接

•在地址栏中键入了新的 URL

•使用前进或后退按钮

•关闭浏览器

•重新加载页面

\*/

# 效果

// $().show([speed,[easing],[fn]]) 显示隐藏的匹配元素。这个就是 'show( speed, [callback] )' 无动画的版本。如果选择的元素是可见的，这个方法将不会改变任何东西。无论这个元素是通过hide()方法隐藏的还是在CSS里设置了display:none;，这个方法都将有效。

/\*

speed:三种预定速度之一的字符串("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

easing:(Optional) 用来指定切换效果，默认是"swing"，可用参数"linear"

fn:在动画完成时执行的函数，每个元素执行一次。

\*/

// $().hide([speed,[easing],[fn]])

$("#btn1").on("click",function(){

$("#box1").show("normal","swing",function(){

console.log("显示完毕")

})

})

$("#btn2").on("click",function(){

$("#box1").hide("fast","linear",function(){

console.log("隐藏完毕")

})

})

//$().slideDown([speed],[easing],[fn]) 通过高度变化（向下增大）来动态地显示所有匹配的元素，在显示完成后可选地触发一个回调函数。这个动画效果只调整元素的高度，可以使匹配的元素以“滑动”的方式显示出来。在jQuery 1.3中，上下的padding和margin也会有动画，效果更流畅。

/\*

speed:三种预定速度之一的字符串("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

easing:(Optional) 用来指定切换效果，默认是"swing"，可用参数"linear"

fn在动画完成时执行的函数，每个元素执行一次。

\*/

// $().slideUp([speed,[easing],[fn]])

// $().slideToggle([speed],[easing],[fn])

$("#btn3").click(function(){

$("#box2").slideToggle("slow","swing",function(){

console.log("切换成功")

})

})

$("#btn4").click(function(){

$("#box2").slideUp("fast","linear",function(){

console.log("上滑成功")

})

})

$("#btn5").click(function(){

$("#box2").slideDown("normal","swing",function(){

console.log("下滑成功")

})

})

// $().fadeIn([speed],[easing],[fn]) 通过不透明度的变化来实现所有匹配元素的淡入效果，并在动画完成后可选地触发一个回调函数。这个动画只调整元素的不透明度，也就是说所有匹配的元素的高度和宽度不会发生变化。

/\*

speed:三种预定速度之一的字符串("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

easing:(Optional) 用来指定切换效果，默认是"swing"，可用参数"linear"

fn:在动画完成时执行的函数，每个元素执行一次。

\*/

// $().fadeOut([speed],[easing],[fn])

// $().fadeTo([[speed],opacity,[easing],[fn]])

/\*

speed:三种预定速度之一的字符串("slow","normal", or "fast")或表示动画时长的毫秒数值(如：1000)

opacity:一个0至1之间表示透明度的数字。

easing:(Optional) 用来指定切换效果，默认是"swing"，可用参数"linear"

fn:在动画完成时执行的函数，每个元素执行一次。

\*/

// $().fadeToggle([speed,[easing],[fn]])

$("#btn6").click(function(){

$("#box3").fadeIn("fast","swing",function(){

})

})

$("#btn7").click(function(){

$("#box3").fadeOut("slow","linear",function(){

})

})

$("#btn8").click(function(){

$("#box3").fadeTo("normal",0.5,function(){

})

})

$("#btn9").click(function(){

$("#box3").fadeToggle("fast","swing")

})

// $().animate(params,[speed],[easing],[fn])

$("#box3").click(function(){

$("#box3").animate({

left:"+=200"

},1000,"swing",function(){

$(this).text("left: " + $(this).css("left"))

$(this).animate({

top : "+=100"

},1000)

})

})

// $().stop([clearQueue],[jumpToEnd]) 停止所有在指定元素上正在运行的动画。如果队列中有等待执行的动画(并且clearQueue没有设为true)，他们将被马上执行

/\*

clearQueue:如果设置成true，则清空队列。可以立即结束动画。

gotoEnd:让当前正在执行的动画立即完成，并且重设show和hide的原始样式，调用回调函数等

\*/

$("#box2").click(function(){

$("#box3").stop(false,true)

})

// $().delay(duration,[queueName]) 设置一个延时来推迟执行队列中之后的项目。jQuery 1.4新增。用于将队列中的函数延时执行。他既可以推迟动画队列的执行，也可以用于自定义队列

/\*

duration:延时时间，单位：毫秒

queueName:队列名词，默认是Fx，动画队列。

\*/

$("#box3").mouseover(function(){

$(this).animate({left:"+=100"},500).delay(1000).animate({top:"-=50"},500)

})

// $().finish( [queue ] ) 停止当前正在运行的动画，删除所有排队的动画，并完成匹配元素所有的动画。当.finish()在一个元素上被调用，立即停止当前正在运行的动画和所有排队的动画（如果有的话），并且他们的CSS属性设置为它们的目标值（所有动画的目标值）。所有排队的动画将被删除。如果第一个参数提供，该字符串表示的队列中的动画将被停止。.finish()方法和.stop(true, true)很相似，.stop(true, true)将清除队列，并且目前的动画跳转到其最终值。但是，不同的是，.finish() 会导致所有排队的动画的CSS属性跳转到他们的最终值。

$("#box2").mouseover(function(){

$("#box3").finish()

})

// jQuery.fx.off 关闭页面上所有的动画。把这个属性设置为true可以立即关闭所有动画(所有效果会立即执行完毕)。有些情况下可能需要这样，比如：\* 你在配置比较低的电脑上使用jQuery。\* 你的一些用户由于动画效果而遇到了 可访问性问题 当把这个属性设成false之后，可以重新开启所有动画。

// jQuery.fx.off = true;

// jQuery.fx.interval 设置动画的显示帧速。 单位：ms

jQuery.fx.interval = 1

# 工具

// jQuery.extend([deep], 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。

如果不指定target，则给jQuery命名空间本身进行扩展。这有助于插件作者为jQuery增加新方法。 如果第一个参数设置为true，则jQuery返回一个深层次的副本，递归地复制找到的任何对象。否则的话，副本会与原对象共享结构。 未定义的属性将不会被复制，然而从对象的原型继承的属性将会被复制。

target:一个对象，如果附加的对象被传递给这个方法将那么它将接收新的属性，如果它是唯一的参数将扩展jQuery的命名空间。

object1:待合并到第一个对象的对象。

objectN:待合并到第一个对象的对象。

deep:如果设为true，则递归合并。（深度复制 ）

target:待修改对象。

object1:待合并到第一个对象的对象。

objectN:待合并到第一个对象的对象。

\*/

let obj1 = {

name:"tom",

istrue: false,

age : 20

}

let obj2 = {

name : "mike",

istrue : true,

age : 21,

grade : 5

}

let obj = $.extend(obj2,obj1) // 如果有相同的属性，则后面的会覆盖前面的

console.log(obj)

console.log(obj1)

let obj3 = {

name:"tom",

istrue: false,

age : 20,

arr : [3,5,6,6]

}

obj = $.extend(obj1,obj3)

console.log(obj)

obj = $.extend(true,obj1,obj3)

console.log(obj)

// jQuery.grep(array, callback, [invert]) 使用过滤函数过滤数组元素。此函数至少传递两个参数：待过滤数组和过滤函数。过滤函数必须返回 true 以保留元素或 false 以删除元素。array:待过滤数组。callback:此函数将处理数组每个元素。第一个参数为当前元素，第二个参数而元素索引值。此函数应返回一个布尔值。另外，此函数可设置为一个字符串，当设置为字符串时，将视为“lambda-form”（缩写形式？），其中 a 代表数组元素，i 代表元素索引值。如“a > 0”代表“function(a){ return a > 0; }”。invert:如果 "invert" 为 false 或为设置，则函数返回数组中由过滤函数返回 true 的元素，当"invert" 为 true，则返回过滤函数中返回 false 的元素集。

let arr = [3,5,6,1]

arr = $.grep(arr,function(item,index){

return item > 1

})

// jQuery.makeArray(obj) 将类数组对象转换为数组对象。类数组对象有 length 属性，其成员索引为 0 至 length - 1。实际中此函数在 jQuery 中将自动使用而无需特意转换。

let weiArr = {}

weiArr[0] = 0

weiArr[1] = 1

weiArr.length = 2

console.log(weiArr.map)

weiArr = $.makeArray(weiArr)

console.log(weiArr.map)

// jQuery.map(arr|obj,callback) 将一个数组中的元素转换到另一个数组中。作为参数的转换函数会为每个数组元素调用，而且会给这个转换函数传递一个表示被转换的元素作为参数。转换函数可以返回转换后的值、null（删除数组中的项目）或一个包含值的数组，并扩展至原始数组中。

arr = $.map(arr,function(item){

return item + 5

})

console.log(arr)

// jQuery.inArray(value,array,[fromIndex]) 确定第一个参数在数组中的位置，从0开始计数(如果没有找到则返回 -1 )。

console.log($.inArray(8,arr,1))

//$().toArray() 把jQuery集合中所有DOM元素恢复成一个数组。

console.log($("#ul1 li").toArray())

// jQuery.merge(first,second) 合并两个数组 返回的结果会修改第一个数组的内容——第一个数组的元素后面跟着第二个数组的元素。要去除重复项，请使用$.unique()

console.log($.merge([0,3,5],[0,6,7]))

// jQuery.unique(array) 删除数组中重复元素。只处理删除DOM元素数组，而不能处理字符串或者数字数组。

console.log($.unique($("li")))

// jQuery.parseJSON(json) 接受一个JSON字符串，返回解析后的对象。

let j = $.parseJSON('{"name":"tom","age":20}')

console.log(j,j.name)

// jQuery.proxy(function,context)

/\*

返回一个新函数，并且这个函数始终保持了特定的作用域。

当有事件处理函数要附加到元素上，但他们的作用域实际是指向另一个对象时，这个方法最有用了。此外，最妙的是，jQuery能够确保即便你绑定的函数是经过jQuery.proxy()处理过的函数，你依然可以传递原先的函数来准确无误地取消绑定。请参考下面的例子。

function:将要被改变作用域的函数

context:一个object，那个函数的作用域会被设置到这个object上来。

context:一个object，那个函数的作用域会被设置到这个object上来。

name：改变上下文中的函数名(这个函数必须是前一个参数 'context' 对象的属性)

\*/

function fn(){

alert(this.name)

}

let obj4 = {

name : "Tom",

click : function(){

alert(this.name)

}

}

fn = $.proxy(fn,obj4)

// fn()

// $("#box1").click( $.proxy(fn,obj4) )

// 等价于

$("#box1").dblclick( $.proxy(obj4,"click") )

// jQuery.contains(container,contained) 一个DOM节点是否包含另一个DOM节点。

console.log( $.contains($("#ul1")[0],$("#ul1 li:first")[0]) )

// jQuery.type(obj) 检测obj的数据类型。

console.log($.type(null),typeof(null)) // null object

console.log($.type([]),typeof([])) // array object

// jQuery.isArray(obj) 测试对象是否为数组。

//jQuery.isFunction(obj) 测试对象是否为函数。注意：jQuery 1.3以后，在IE浏览器里，浏览器提供的函数比如'alert'还有 DOM 元素的方法比如 'getAttribute' 将不认为是函数

// jQuery.isEmptyObject(obj) jQuery 1.4 新增。测试对象是否是空对象（不包含任何属性）。jQuery 1.4 中，这个方法既检测对象本身的属性，也检测从原型继承的属性（因此没有使用hasOwnProperty）。

function Super(){

this.name = "tom"

}

function Sub(){

}

Sub.prototype = new Super()

let sub = new Sub()

console.log($.isEmptyObject(sub)) // false

console.log($.isEmptyObject({})) // true

// jQuery.isNumeric(value) 确定它的参数是否是一个数字。$.isNumeric() 方法检查它的参数是否代表一个数值。如果是这样，它返回 true。否则，它返回false。该参数可以是任何类型的

console.log("-------------------------")

console.log( $.isNumeric("") ) // false

console.log( $.isNumeric("22") ) //true

console.log( $.isNumeric(22) ) //true

console.log( $.isNumeric(null) ) // false

console.log( $.isNumeric(NaN) ) // false

// jQuery.trim(str) 去掉字符串起始和结尾的空格。

console.log("----"+ $.trim(" trim ") +"---")

// jQuery.error(message) 接受一个字符串，并且直接抛出一个包含这个字符串的异常。这个方法的主要目的是提供给插件开发人员，让他们可以重载这个方法，并以更好的方式显示错误，或者提供更多信息。

// $.fn.jquery 代表 jQuery 版本号的字符串。.jquery 属性是通过 jQuery 原型赋值的，通过使用它的别名 $.fn 进行引用。它是一个含有 jQuery 版本号的字符串，例如 "1.5.0" 或 "1.4.4".

console.log($("#ul1").jquery,$.fn.jquery)

// 可以用来判断一个对象是否是jQuery对象

if(document.getElementById("ul1").jquery){

console.log("这是一个jQuery对象")

}else{

console.log("这不是一个jQuery对象")

}

# 事件对象

// event.currentTarget 在事件冒泡阶段中的当前DOM元素 currentTarget的匹配`this`关键字。就是绑定事件的那个对象

// event.target 最初触发事件的DOM元素。 这是注册事件时的对象，或者它的子元素。通常用于比较 event.target 和 this 来确定事件是不是由于冒泡而触发的。经常用于事件冒泡时处理事件委托。 真正触发事件的那个元素，不一定是绑定事件时的那个元素，可能是他的子元素

// event.relatedTarget 在事件中涉及的其它任何DOM元素。对于 mouseout 事件，它指向被进入的元素；对于 mousein 事件，它指向被离开的元素。

// $("#box").on("click",function(ev){

// console.log(ev.currentTarget)

// console.log(ev.currentTarget === this) // true

// console.log(ev.target)

// console.log(ev.target === this)

// })

// event.data 当前执行的处理器被绑定的时候，包含可选的数据传递给jQuery.fn.bind。

// $("#btn1").on("click",{name : "Tom"},function(ev){

// console.log(ev.data.name)

// })

// event.isDefaultPrevented() 根据事件对象中是否调用过 event.preventDefault() 方法来返回一个布尔值。

$("#btn1").on("click",function(ev){

console.log(ev.namespace)

console.log(ev.isDefaultPrevented())

ev.preventDefault()

console.log(ev.isDefaultPrevented())

})

// event.isImmediatePropagationStopped() 根据事件对象中是否调用过 event.stopImmediatePropagation() 方法来返回一个布尔值。

// event.isPropagationStopped() 根据事件对象中是否调用过 event.stopPropagation() 方法来返回一个布尔值。

// event.pageX 鼠标相对于文档的左边缘的位置。

// event.pageY 鼠标相对于文档的上边缘的位置。

// event.preventDefault() 阻止默认事件行为的触发。

// event.stopImmediatePropagation() 阻止剩余的事件处理函数执行并且防止事件冒泡到DOM树上。除了阻止元素上其它的事件处理函数的执行，这个方法还会通过在内部调用 event.stopPropagation() 来停止事件冒泡。如果仅仅想要停止事件冒泡到前辈元素上，而让这个元素上的其它事件处理函数继续执行，我们可以使用event.stopPropagation() 来代替。

// event.stopPropagation() 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。我们可以用 event.isPropagationStopped() 来确定这个方法是否(在那个事件对象上)调用过了。这个方法对 trigger() 来自定义的事件同样有效。注意，这不会阻止同一个元素上的其它事件处理函数的运行

// event.timeStamp 这个属性返回事件触发时距离1970年1月1日的毫秒数。这可以很方便的检测某个jQuery函数的性能。

// event.type 返回触发的事件类型

// event.which 针对键盘和鼠标事件，这个属性能确定你到底按的是哪个键或按钮。event.which 将 event.keyCode 和 event.charCode 标准化了。推荐用 event.which 来监视键盘输入。

# 回调函数

let callbacks = $.Callbacks()

function fn1(value){

console.log(this + " , " +value)

}

function fn2(value){

console.log("fn2 : " + value)

}

// callbacks.add(callbacks) 向回调列表中添加一个回调或回调集合

console.log("----------------callbacks.add(callbacks)-------------------")

callbacks.add(fn1)

callbacks.fire("hello")

callbacks.add(fn2)

callbacks.fire("world")

// callbacks.disable() 禁用回调列表中的回调

console.log("----------------callbacks.disable()-------------------")

// callbacks.disable()

callbacks.fire("fire") // 无反应

callbacks.add(fn1)

callbacks.fire("fire") // 无反应

// callbacks.empty() 从列表中删除所有的回调. 如果前面callback被disable了，那么这个方法返回的也是false

console.log("----------------callbacks.empty()-------------------")

console.log(callbacks.has(fn1),callbacks.has(fn2))

callbacks.empty()

console.log(callbacks.has(fn1),callbacks.has(fn2))

// callbacks.fire(arguments)

// callbacks.fired() 测试callbacks是否fire过

console.log("----------------callbacks.fired()-------------------")

console.log(callbacks.fired())

let callbacks2 = $.Callbacks()

console.log(callbacks2.fired())

// callbacks.fireWith(obj,args) 以给定的obj为this，给定的args为参数，触发回调函数 args为数组形式

console.log("----------------callbacks.fireWith(obj,args)-------------------")

callbacks.add(fn1)

callbacks.fireWith(window,["window"])

// callbacks.has(callback) 判断回调列表中是否含有指定的回调函数

console.log("----------------callbacks.has(callback)-------------------")

console.log(callbacks.has(fn1),callbacks.has(fn2))

// callbacks.lock() 锁定在其当前状态的回调列表。

console.log("----------------callbacks.lock()-------------------")

callbacks.lock()

callbacks.fire("fn1")

callbacks.add(fn2) // 无效

callbacks.fire("fn2") // 无效

// callbacks.locked() 确定回调列表是否被锁定

console.log("----------------callbacks.locked()-------------------")

console.log(callbacks.locked())

// callbacks.remove(callback) 删除回调列表中的指定回调函数

console.log("----------------callbacks.remove(callback)-------------------")

callbacks2.add(fn1)

callbacks2.fire("fn1")

callbacks2.remove(fn1)

callbacks2.fire("fn1") // 无响应

console.log(callbacks2.has(fn1)) // false

// $.Callbacks(flags)

/\*

flags:

once: will ensure the callback list can only be fired once (like a Deferred)

\* 确保这个callback列表只会被执行一次

\*

\* memory: will keep track of previous values and will call any callback added

\* after the list has been fired right away with the latest "memorized"

\* values (like a Deferred)

\* 会记住前一次执行时的参数，执行后被添加的callback会被立即执行，并且传入被记住的之前的参数

\*

\* unique: will ensure a callback can only be added once (no duplicate in the list)

\* 确保一个callback之会被添加一次

\*

\* stopOnFalse: interrupt callings when a callback returns false

\* 当某个callback返回false时，终止执行callback列表

\*/

// once

console.log("----------------once-------------------")

let callbacks3 = $.Callbacks("once")

callbacks3.add(fn1)

callbacks3.fire("once")

callbacks3.fire("once") // 无反应

callbacks3.add(fn2)

callbacks3.fire("once2") // 无反应

// memory

console.log("----------------memory-------------------")

callbacks3 = $.Callbacks("memory")

callbacks3.add(fn1)

callbacks3.fire("memory")

callbacks3.add(fn2) // fn2直接执行

// callbacks.lock() 过的，如果以memory为参数创建的callback，则添加新回调函数时仍然可以执行

callbacks3.lock()

callbacks3.add(fn1) // fn1 可以执行

callbacks3.disable()

callbacks3.add(fn2) // fn2 不执行

// unique

console.log("----------------unique-------------------")

callbacks3 = $.Callbacks("unique")

callbacks3.add(fn1)

callbacks3.add(fn1)

callbacks3.fire("unique")

callbacks = $.Callbacks()

callbacks.add(fn2)

callbacks.add(fn2)

callbacks.fire("no unique") // fn2 执行两次，因为回调队列中有两个fn2

// stopOnFalse

console.log("----------------stopOnFalse-------------------")

function fn3(value){

console.log("fn3 : " + value)

return false

}

callbacks = $.Callbacks("stopOnFalse")

callbacks.add(fn1)

callbacks.add(fn3)

callbacks.add(fn2)

callbacks.fire("stopOnFalse") // fn2 不执行，因为fn3中返回了false

# Ajax

<label>label</label>

<button>发起ajax请求</button>

<form action="#">

<input type="text" name="name" value="" placeholder="your name">

<input type="email" name="email" placeholder="your name">

<input type="checkbox" name="hobby" value="basketball">

<input type="checkbox" name="hobby" value="football">

<input type="checkbox" name="hobby" value="baseball">

<input type="submit" name="submit" value="submit" id="submit">

</form>

<script type="text/javascript" src="jquery-1.12.4.js"></script>

<script type="text/javascript">

// jQuery.ajax([settings])

/\*

settings

可选。用于配置 Ajax 请求的键值对集合。

可以通过 $.ajaxSetup() 设置任何选项的默认值

settings:选项

accepts Map

默认： 取决于数据类型。

内容类型发送请求头，告诉服务器什么样的响应会接受返回。如果accepts设置需要修改，推荐在$.ajaxSetup()方法中做一次。

async Boolean

(默认: true) 默认设置下，所有请求均为异步请求。如果需要发送同步请求，请将此选项设置为 false。注意，同步请求将锁住浏览器，用户其它操作必须等待请求完成才可以执行。

beforeSend(XHR) Function

发送请求前可修改 XMLHttpRequest 对象的函数，如添加自定义 HTTP 头。XMLHttpRequest 对象是唯一的参数。这是一个 Ajax 事件。如果返回false可以取消本次ajax请求。

function (XMLHttpRequest) {

this; // 调用本次AJAX请求时传递的options参数

}

cache Boolean

(默认: true,dataType为script和jsonp时默认为false) jQuery 1.2 新功能，设置为 false 将不缓存此页面。

complete(XHR, TS) Function

请求完成后回调函数 (请求成功或失败之后均调用)。参数： XMLHttpRequest 对象和一个描述成功请求类型的字符串。 Ajax 事件。

function (XMLHttpRequest, textStatus) {

this; // 调用本次AJAX请求时传递的options参数

}

contents MapV 1.5

一个以"{字符串:正则表达式}"配对的对象，用来确定jQuery将如何解析响应，给定其内容类型。

contentTypeString

(默认: "application/x-www-form-urlencoded") 发送信息至服务器时内容编码类型。默认值适合大多数情况。如果你明确地传递了一个content-type给 $.ajax() 那么他必定会发送给服务器（即使没有数据要发送）

context Object

这个对象用于设置Ajax相关回调函数的上下文。也就是说，让回调函数内this指向这个对象（如果不设定这个参数，那么this就指向调用本次AJAX请求时传递的options参数）。比如指定一个DOM元素作为context参数，这样就设置了success回调函数的上下文为这个DOM元素。就像这样：

$.ajax({ url: "test.html", context: document.body, success: function(){

$(this).addClass("done");

}});

converters map V1.5

默认： {"\* text": window.String, "text html": true, "text json": jQuery.parseJSON, "text xml": jQuery.parseXML}

一个数据类型对数据类型转换器的对象。每个转换器的值是一个函数，返回响应的转化值

crossDomain mapV1.5

默认： 同域请求为false

跨域请求为true如果你想强制跨域请求（如JSONP形式）同一域，设置crossDomain为true。这使得例如，服务器端重定向到另一个域

data Object,String

发送到服务器的数据。将自动转换为请求字符串格式。GET 请求中将附加在 URL 后。查看 processData 选项说明以禁止此自动转换。必须为 Key/Value 格式。如果为数组，jQuery 将自动为不同值对应同一个名称。如 {foo:["bar1", "bar2"]} 转换为 "&foo=bar1&foo=bar2"。

dataFilter Function

给Ajax返回的原始数据的进行预处理的函数。提供data和type两个参数：data是Ajax返回的原始数据，type是调用jQuery.ajax时提供的dataType参数。函数返回的值将由jQuery进一步处理。

function (data, type) {

// 对Ajax返回的原始数据进行预处理

return data // 返回处理后的数据

}

dataType String

预期服务器返回的数据类型。如果不指定，jQuery 将自动根据 HTTP 包 MIME 信息来智能判断，比如XML MIME类型就被识别为XML。在1.4中，JSON就会生成一个JavaScript对象，而script则会执行这个脚本。随后服务器端返回的数据会根据这个值解析后，传递给回调函数。可用值:

"xml": 返回 XML 文档，可用 jQuery 处理。

"html": 返回纯文本 HTML 信息；包含的script标签会在插入dom时执行。

"script": 返回纯文本 JavaScript 代码。不会自动缓存结果。除非设置了"cache"参数。'''注意：'''在远程请求时(不在同一个域下)，所有POST请求都将转为GET请求。(因为将使用DOM的script标签来加载)

"json": 返回 JSON 数据 。

"jsonp": JSONP 格式。使用 JSONP 形式调用函数时，如 "myurl?callback=?" jQuery 将自动替换 ? 为正确的函数名，以执行回调函数。

"text": 返回纯文本字符串

error Function

(默认: 自动判断 (xml 或 html)) 请求失败时调用此函数。有以下三个参数：XMLHttpRequest 对象、错误信息、（可选）捕获的异常对象。如果发生了错误，错误信息（第二个参数）除了得到null之外，还可能是"timeout", "error", "notmodified" 和 "parsererror"。Ajax 事件。

function (XMLHttpRequest, textStatus, errorThrown) {

// 通常 textStatus 和 errorThrown 之中

// 只有一个会包含信息

this; // 调用本次AJAX请求时传递的options参数

}

global Boolean

(默认: true) 是否触发全局 AJAX 事件。设置为 false 将不会触发全局 AJAX 事件，如 ajaxStart 或 ajaxStop 可用于控制不同的 Ajax 事件。

headersmapV1.5

Default: {}

一个额外的"{键:值}"对映射到请求一起发送。此设置被设置之前beforeSend函数被调用;因此，消息头中的值设置可以在覆盖beforeSend函数范围内的任何设置。

ifModified Boolean

(默认: false) 仅在服务器数据改变时获取新数据。使用 HTTP 包 Last-Modified 头信息判断。在jQuery 1.4中，他也会检查服务器指定的'etag'来确定数据没有被修改过。

isLocal map V1.5.1

默认: 取决于当前的位置协议

允许当前环境被认定为“本地”，（如文件系统），即使jQuery默认情况下不会承认它。以下协议目前公认为本地：file, \*-extension, and widget。如果isLocal设置需要修改，建议在$.ajaxSetup()方法中这样做一次。

jsonp String

在一个jsonp请求中重写回调函数的名字。这个值用来替代在"callback=?"这种GET或POST请求中URL参数里的"callback"部分，比如{jsonp:'onJsonPLoad'}会导致将"onJsonPLoad=?"传给服务器。

jsonpCallback String

为jsonp请求指定一个回调函数名。这个值将用来取代jQuery自动生成的随机函数名。这主要用来让jQuery生成度独特的函数名，这样管理请求更容易，也能方便地提供回调函数和错误处理。你也可以在想让浏览器缓存GET请求的时候，指定这个回调函数名。

mimeType String V1.5.1

一个mime类型用来覆盖XHR的 MIME类型。

password String

用于响应HTTP访问认证请求的密码

processData Boolean

(默认: true) 默认情况下，通过data选项传递进来的数据，如果是一个对象(技术上讲只要不是字符串)，都会处理转化成一个查询字符串，以配合默认内容类型 "application/x-www-form-urlencoded"。如果要发送 DOM 树信息或其它不希望转换的信息，请设置为 false。

scriptCharset String

只有当请求时dataType为"jsonp"或"script"，并且type是"GET"才会用于强制修改charset。通常只在本地和远程的内容编码不同时使用。

statusCode map V1.5

默认: {}

一组数值的HTTP代码和函数对象，当响应时调用了相应的代码。例如，如果响应状态是404，将触发以下警报：

$.ajax({

statusCode: {404: function() {

alert('page not found');

}

});

success(data, textStatus, jqXHR) Function,Array

请求成功后的回调函数。参数：由服务器返回，并根据dataType参数进行处理后的数据；描述状态的字符串。还有 jqXHR（在jQuery 1.4.x的中，XMLHttpRequest） 对象 。在jQuery 1.5， 成功设置可以接受一个函数数组。每个函数将被依次调用。 Ajax 事件。

function (data, textStatus) {

// data 可能是 xmlDoc, jsonObj, html, text, 等等...

this; // 调用本次AJAX请求时传递的options参数

}

traditional Boolean

如果你想要用传统的方式来序列化数据，那么就设置为true。请参考工具分类下面的jQuery.param 方法。

timeout Number

设置请求超时时间（毫秒）。此设置将覆盖全局设置。

type String

(默认: "GET") 请求方式 ("POST" 或 "GET")， 默认为 "GET"。注意：其它 HTTP 请求方法，如 PUT 和 DELETE 也可以使用，但仅部分浏览器支持。

urlString

(默认: 当前页地址) 发送请求的地址。

username String

用于响应HTTP访问认证请求的用户名

xhr Function

需要返回一个XMLHttpRequest 对象。默认在IE下是ActiveXObject 而其他情况下是XMLHttpRequest 。用于重写或者提供一个增强的XMLHttpRequest 对象。这个参数在jQuery 1.3以前不可用。

xhrFields map V1.5

一对“文件名-文件值”在本机设置XHR对象。例如，如果需要的话，你可以用它来设置withCredentials为true的跨域请求。

\*/

// let xhr = $.ajax({

// type : "GET",

// url : "web workers.html",

// data : "",

// // async : false

// success : function(data,textStatus,jqXHR){

// // $(document.body).text(data)

// }

// })

// console.log(xhr.responseText)

// $().load(url,data,function(response,status,xhr))

/\*

url 规定要将请求发送到哪个 URL。

data 可选。规定连同请求发送到服务器的数据。

function(response,status,xhr)

可选。规定当请求完成时运行的函数。

额外的参数：

response - 包含来自请求的结果数据

status - 包含请求的状态（"success", "notmodified", "error", "timeout" 或 "parsererror"）

xhr - 包含 XMLHttpRequest 对象

详细说明

该方法是最简单的从服务器获取数据的方法。它几乎与 $.get(url, data, success) 等价，不同的是它不是全局函数，并且它拥有隐式的回调函数。当侦测到成功的响应时（比如，当 textStatus 为 "success" 或 "notmodified" 时），.load() 将匹配元素的 HTML 内容设置为返回的数据。这意味着该方法的大多数使用会非常简单：

$("#result").load("ajax/test.html");

如果提供回调函数，则会在执行 post-processing 之后执行该函数：

$("#result").load("ajax/test.html", function() {

alert("Load was performed.");

});

上面的两个例子中，如果当前文档不包含 "result" ID，则不会执行 .load() 方法。

如果提供的数据是对象，则使用 POST 方法；否则使用 GET 方法。

加载页面片段

.load() 方法，与 $.get() 不同，允许我们规定要插入的远程文档的某个部分。这一点是通过 url 参数的特殊语法实现的。如果该字符串中包含一个或多个空格，紧接第一个空格的字符串则是决定所加载内容的 jQuery 选择器。

我们可以修改上面的例子，这样就可以使用所获得文档的某部分：

$("#result").load("ajax/test.html #container");

如果执行该方法，则会取回 ajax/test.html 的内容，不过然后，jQuery 会解析被返回的文档，来查找带有容器 ID 的元素。该元素，连同其内容，会被插入带有结果 ID 的元素中，所取回文档的其余部分会被丢弃。

jQuery 使用浏览器的 .innerHTML 属性来解析被取回的文档，并把它插入当前文档。在此过程中，浏览器常会从文档中过滤掉元素，比如 <html>, <title> 或 <head> 元素。结果是，由 .load() 取回的元素可能与由浏览器直接取回的文档不完全相同。

注释：由于浏览器安全方面的限制，大多数 "Ajax" 请求遵守同源策略；请求无法从不同的域、子域或协议成功地取回数据。

\*/

// $("label").load("1.txt")

// $("label").load("1.txt",function(){

// alert("加载成功")

// })

// $("label").load("1.html #contents p")

// $.get(url,data,success(response,status,xhr),dataType)

/\*

get() 方法通过远程 HTTP GET 请求载入信息。

这是一个简单的 GET 请求功能以取代复杂 $.ajax 。请求成功时可调用回调函数。如果需要在出错时执行函数，请使用 $.ajax。

url 必需。规定将请求发送的哪个 URL。

data 可选。规定连同请求发送到服务器的数据。

success(response,status,xhr)

可选。规定当请求成功时运行的函数。

额外的参数：

response - 包含来自请求的结果数据

status - 包含请求的状态

xhr - 包含 XMLHttpRequest 对象

dataType

可选。规定预计的服务器响应的数据类型。

默认地，jQuery 将智能判断。如果传入不存在的数据类型的参数，则不会有返回值，如全是文本，但传入了 json ，则没有返回值

可能的类型：

"xml"

"html"

"text"

"script"

"json"

"jsonp"

该函数是简写的 Ajax 函数，等价于：

$.ajax({

url: url,

data: data,

success: success,

dataType: dataType

});

根据响应的不同的 MIME 类型，传递给 success 回调函数的返回数据也有所不同，这些数据可以是 XML root 元素、文本字符串、JavaScript 文件或者 JSON 对象。也可向 success 回调函数传递响应的文本状态。

对于 jQuery 1.4，也可以向 success 回调函数传递 XMLHttpRequest 对象。

\*/

// $.get("1.txt",function(data,status,xhr){

// console.log(data)

// })

// $.post(url,data,success(data, textStatus, jqXHR),dataType)

/\*

post() 方法通过 HTTP POST 请求从服务器载入数据。

url 必需。规定把请求发送到哪个 URL。

data 可选。映射或字符串值。规定连同请求发送到服务器的数据。

success(data, textStatus, jqXHR) 可选。请求成功时执行的回调函数。

dataType

可选。规定预期的服务器响应的数据类型。

默认执行智能判断（xml、json、script 或 html）。

该函数是简写的 Ajax 函数，等价于：

$.ajax({

type: 'POST',

url: url,

data: data,

success: success,

dataType: dataType

});

根据响应的不同的 MIME 类型，传递给 success 回调函数的返回数据也有所不同，这些数据可以是 XML 根元素、文本字符串、JavaScript 文件或者 JSON 对象。也可向 success 回调函数传递响应的文本状态。

对于 jQuery 1.5，也可以向 success 回调函数传递 jqXHR 对象（jQuery 1.4 中传递的是 XMLHttpRequest 对象）。

大部分实现会规定一个 success 函数：

$.post("ajax/test.html", function(data) {

$(".result").html(data);

});

本例读取被请求的 HTML 片段，并插入页面中。

通过 POST 读取的页面不被缓存，因此 jQuery.ajaxSetup() 中的 cache 和 ifModified 选项不会影响这些请求。

注释：由于浏览器安全方面的限制，大多数 "Ajax" 请求遵守同源策略；请求无法从不同的域、子域或协议成功地取回数据。

注释：如果由 jQuery.post() 发起的请求返回错误代码，那么不会有任何提示，除非脚本已调用了全局的 .ajaxError() 方法。或者对于 jQuery 1.5，jQuery.post() 返回的 jqXHR 对象的 .error() 方法也可以用于错误处理。

\*/

// $.post("1.html",{name : "tom"},function(data,status,xhr){

// $("label").text(data)

// })

// $.getJSON(url,data,success(data,status,xhr))

/\*

通过 HTTP GET 请求载入 JSON 数据。

在 jQuery 1.2 中，您可以通过使用 JSONP 形式的回调函数来加载其他网域的 JSON 数据，如 "myurl?callback=?"。jQuery 将自动替换 ? 为正确的函数名，以执行回调函数。 注意：此行以后的代码将在这个回调函数执行前执行。

url 必需。规定将请求发送的哪个 URL。

data 可选。规定连同请求发送到服务器的数据。

success(data,status,xhr)

可选。规定当请求成功时运行的函数。

额外的参数：

data - 包含来自请求的结果数据

status - 包含请求的状态

xhr - 包含 XMLHttpRequest 对象

该函数是简写的 Ajax 函数，等价于：

$.ajax({

url: url,

data: data,

success: callback,

dataType: json

});

发送到服务器的数据可作为查询字符串附加到 URL 之后。如果 data 参数的值是对象（映射），那么在附加到 URL 之前将转换为字符串，并进行 URL 编码。

传递给 callback 的返回数据，可以是 JavaScript 对象，或以 JSON 结构定义的数组，并使用 $.parseJSON() 方法进行解析。

\*/

// $.getJSON("json.js",function(json,status,xhr){

// $.each(json,function(key,value){

// console.log(key + " : " + value)

// })

// })

// $.getScript(url,success(response,status))

/\*

getScript() 方法通过 HTTP GET 请求载入并执行 JavaScript 文件。

url 将要请求的 URL 字符串。

success(response,status)

可选。规定请求成功后执行的回调函数。

额外的参数：

response - 包含来自请求的结果数据

status - 包含请求的状态（"success", "notmodified", "error", "timeout" 或 "parsererror"）

该函数是简写的 Ajax 函数，等价于：

$.ajax({

url: url,

dataType: "script",

success: success

});

这里的回调函数会传入返回的 JavaScript 文件。这通常不怎么有用，因为那时脚本已经运行了。

载入的脚本在全局环境中执行，因此能够引用其他变量，并使用 jQuery 函数。

比如加载一个 test.js 文件，里边包含下面这段代码：

$(".result").html("<p>Lorem ipsum dolor sit amet.</p>");

通过引用该文件名，就可以载入并运行这段脚本：

$.getScript("ajax/test.js", function() {

alert("Load was performed.");

});

注释：jQuery 1.2 版本之前，getScript 只能调用同域 JS 文件。 1.2中，您可以跨域调用 JavaScript 文件。注意：Safari 2 或更早的版本不能在全局作用域中同步执行脚本。如果通过 getScript 加入脚本，请加入延时函数。

\*/

// $.getScript("script.js",function(response,status){

// alert("script加载完成")

// })

// $()..ajaxStart(function(event,xhr,options))

/\*

ajaxStart() 方法在 AJAX 请求发送前执行函数。它是一个 Ajax 事件。

无论在何时发送 Ajax 请求，jQuery 都会检查是否存在其他 Ajax 请求。如果不存在，则 jQuery 会触发该 ajaxStart 事件。在此时，由 .ajaxStart() 方法注册的任何函数都会被执行。

function(event,xhr,options)

必需。规定当请求开始时运行的函数。

额外的参数：

event - 包含 event 对象

xhr - 包含 XMLHttpRequest 对象

options - 包含 AJAX 请求中使用的选项

\*/

$("button").click(function(){

$("label").load("1.txt")

})

$("label").ajaxSend(function(){

alert(1)

console.log("ajax start")

alert(2)

})

// console.log($("label").ajaxStart)

// $().ajaxSend([function(event,xhr,options)]) AJAX 请求发送前执行函数。Ajax 事件。

// $().ajaxSuccess(function(event,xhr,options)) ajaxSuccess() 方法在 AJAX 请求成功时执行函数。它是一个 Ajax 事件。XMLHttpRequest 对象和设置作为参数传递给回调函数。无论 Ajax 请求在何时成功完成 ，jQuery 都会触发该 ajaxSuccess 事件。在此时，由 .ajaxSuccess() 方法注册的任何函数都会被执行。

// $().ajaxComplete(function(event,xhr,options)) ajaxComplete() 方法在 AJAX 请求完成时执行函数。它是一个 Ajax 事件。与 ajaxSuccess() 不同，通过 ajaxComplete() 方法规定的函数会在请求完成时运行，即使请求并未成功。

// $().ajaxError() 方法在 AJAX 请求发生错误时执行函数。它是一个 Ajax 事件。

/\*

function(event,xhr,options,exc)

必需。规定当请求失败时运行的函数。

额外的参数：

event - 包含 event 对象

xhr - 包含 XMLHttpRequest 对象

options - 包含 AJAX 请求中使用的选项

exc - 包含 JavaScript exception

\*/

// $().ajaxStop(function()) ajaxStop() 方法在 AJAX 请求结束时执行函数。它是一个 Ajax 事件。无论 Ajax 请求在何时完成 ，jQuery 都会检查是否存在其他 Ajax 请求。如果不存在，则 jQuery 会触发该 ajaxStop 事件。在此时，由 .ajaxStop() 方法注册的任何函数都会被执行。

// $.param(object,traditional)

/\*

param() 方法创建数组或对象的序列化表示。

该序列化值可在进行 AJAX 请求时在 URL 查询字符串中使用。

object 要进行序列化的数组或对象。

traditional 规定是否使用传统的方式浅层进行序列化（参数序列化）。

\*/

let person = {

name : "小明",

age : 20,

sex : "male"

}

let encode = $.param(person)

let decode = decodeURIComponent(encode)

console.log(encode) // name=%E5%B0%8F%E6%98%8E&age=20&sex=male

console.log(decode) // name=小明&age=20&sex=male

// $(selector).serialize()

/\*

serialize() 方法通过序列化表单值，创建 URL 编码文本字符串。

您可以选择一个或多个表单元素（比如 input 及/或 文本框），或者 form 元素本身。

序列化的值可在生成 AJAX 请求时用于 URL 查询字符串中。

.serialize() 方法可以操作已选取个别表单元素的 jQuery 对象，比如 <input>, <textarea> 以及 <select>。不过，选择 <form> 标签本身进行序列化一般更容易些.

只会将”成功的控件“序列化为字符串。如果不使用按钮来提交表单，则不对提交按钮的值序列化。如果要表单元素的值包含到序列字符串中，元素必须使用 name 属性。

\*/

// $(selector).serializeArray() serializeArray() 方法通过序列化表单值来创建对象数组（名称和值）。您可以选择一个或多个表单元素（比如 input 及/或 textarea），或者 form 元素本身。

$("#submit").click(function(){

console.log($("form").serialize()) // name=aaaaaaaaa&email=222222222222%40qq.com&hobby=basketball&hobby=football

console.log($("form").serializeArray())

/\*

(4) [{…}, {…}, {…}, {…}]

0:{name: "name", value: "aaaaaaaaa"}

1:{name: "email", value: "222222222222@qq.com"}

2:{name: "hobby", value: "basketball"}

3:{name: "hobby", value: "football"}

length:4

\_\_proto\_\_:Array(0)

\*/

return false

})