

jQuery---源码 (0.4)

2014年8月27日 10:23

<http://www.cnblogs.com/chyingp/archive/2013/06/03/jquery-souce-code-study.html>
<http://www.cnblogs.com/nuysoft/archive/2011/11/14/2248023.html>

jquery2.0和1.0的区别是：2.0将不支持IE6/7/8。

<http://www.cnblogs.com/nuysoft/>
<http://nuysoft.iteye.com/>

jquery源码阅读，这位作者出了一本书叫jquery技术内幕
<http://item.jd.com/1080321026.html>

什么是JQ？

一个优秀的JS库，大型开发必备

JQ好处？

- 简化JS的复杂操作
- 不需要关心兼容性
- 提供大量实用方法

最前面的（87）代表源码中的行数

```
=====
miaov1~3
版本2.0.3
结构：
(function(){
    ( 21,94 ) 定义了一些变量和函数，重要函数：jQuery = function( selector, context ){
    ( 96, 280 ) 给jq对象添加方法和属性
    ( 285, 347 ) extend：JQ的继承方法
    ( 349, 817 ) jQuery.extend()：扩展工具方法，比如$.trim();工具方法既可以给jquery对象用，也可以给原生js用
    (877, 2856) sizzle 复杂元素选择器的实现
    ( 2880, 3042 ) Callbacks：回调对象：作用是函数的统一管理
    ( 3043, 3183 ) Deferred：延迟对象：作用是对异步的统一管理
    ( 3184, 3295 ) support：功能检测
    ( 3308, 3652 ) data()：数据缓存，避免大数据添加到元素身上 造成内存泄露
    ( 3653, 3797 ) queue()：队列管理
    ( 3803, 4299 ) attr() prop() val() addClass()等，对元素属性的操作
    ( 4300, 5128 ) on() trigger()：事件操作的相关方法
    (5140, 6057) DOM操作方法：添加 删除 获取 包装 DOM筛选
    (6058, 6620) css()：样式的操作
    (6621, 7854) 提交的数据和ajax()：ajax() load() getJson()
    (7855, 8584) animate()：运动的方法
    (8585, 8792) offset()：位置和尺寸的方法
    (8804, 8821) JQ支持模块化的模式
    (8826) window.jQuery = window.$ = jQuery; 通过外面找到jQuery，对外提供的接口
})();
( 21,94 ) =====
```

(21,94) 定义了一些变量和函数，重要函数：jQuery = function(selector, context){ miaov4~6，

	<pre>/*! * jQuery JavaScript Library v2.0.3 * http://jquery.com/ * * Includes Sizzle.js * http://sizzlejs.com/ * * Copyright 2005, 2013 jQuery Foundation, Inc. and other contributors * Released under the MIT license * http://jquery.org/license * * Date: 2013-07-03T13:30Z */</pre>
<p>(14) 为什么使用window？</p> <pre>(function(){ window })(window);</pre> <p>1.使用局部变量查找速度快 2.传参可以进行压缩</p> <p>为什么传参undefined？</p> <p>防止undefined被修改</p>	<pre>(function(window, undefined) {</pre>
<p>(20) "use strict"：不建议去掉注释，去掉会造成假死状态等。目前此bug已修复</p>	<pre>// Can't do this because several apps including ASP.NET trace // the stack via arguments.caller.caller and Firefox dies if // you try to trace through "use strict" call chains. (#13335) // Support: Firefox 18+ // "use strict";</pre>
<p>(23) rootjQuery：</p> <p>1.方便压缩；2.定义一个变量方便后期可维护</p>	<pre>var // A central reference to the root jQuery(document) rootjQuery,</pre>
<p>(26) readyList：</p> <p>跟DOM加载有关</p>	<pre>// The deferred used on DOM ready readyList,</pre>
<p>(30) core_strundefined：得到的是字符串形式的undefined</p> <pre>window.a == undefined; typeof window.a == 'undefined';</pre> <p>一般情况下，以上两种方式判断是ok的。</p> <p>但是在老版本浏览器下IE9，当判断的是一个xmlNode时，判断不出来。所以最好采用typeof的方法进行判断。</p>	<pre>// Support: IE9 // For `typeof xmlNode.method` instead of `xmlNode.method !== undefined` core_strundefined = typeof undefined,</pre>
<p>(33) 对window下的一些变量进行存储，方便压缩</p>	<pre>// Use the correct document accordingly with window argument (sandbox) location = window.location,</pre>

	<pre>document = window.document, docElem = document.documentElement,</pre>
(38 , 41) 防冲突 , 如果在引入jQuery库之前已经定义了jQuery或者\$就先暂存起来	<pre>// Map over jQuery in case of overwrite jQuery = window.jQuery, // Map over the \$ in case of overwrite \$ = window.\$,</pre>
(44) 使用\$.type()时用到的 , 它里面可能会存成这样的	<pre>// [[Class]] -> type pairs class2type = {},</pre>
(47) 没有什么实际用处了 , 老版本中是和数据缓存有关的	<pre>// List of deleted data cache ids, so we can reuse them core_deletedIds = [],</pre>
(49) 版本号	<pre>core_version = "2.0.3",</pre>
(52,58) 存一些方法名	<pre>// Save a reference to some core methods core_concat = core_deletedIds.concat, core_push = core_deletedIds.push, core_slice = core_deletedIds.slice, core_indexOf = core_deletedIds.indexOf, core_toString = class2type.toString, core_hasOwn = class2type.hasOwnProperty, core_trim = core_version.trim,</pre>
(61) jQuery方法, fn就是prototype , jQuery.fn.init是一个构造函数 <pre>function Aaa(){ Aaa.prototype.init = function(){}; Aaa.prototype.css = function(){}; var a1 = new Aaa(); a1.init(); a1.css(); 下面的与上面的不同, 可以一目了然的找到css, 不用做烦琐的初始化工作 function jQuery(){ return new jQuery.prototype.init(); } jQuery.prototype.init = function(){}; jQuery.prototype.css = function(){}; jQuery.prototype.init.prototype = jQuery.prototype; jQuery().css(); 283行有 : jQuery.fn.init.prototype = jQuery.prototype; 故在 jQuery.prototype下添加的任何方法, 可以通过jQuery.prototype.init new出来的对象使用</pre>	<pre>// Define a local copy of jQuery jQuery = function(selector, context) { // The jQuery object is actually just the init constructor 'enhanced' return new jQuery.fn.init(selector, context, rootjQuery); },</pre>
(67) 一些正则 , 在css方法时使用 <pre>core_pnum 匹配数字 core_rnotwhite 匹配单词 rquickExpr 匹配标签和id, 通过location.hash防止XSS注入 <p>aaa 或 #id rsingleTag 匹配独立的空标签 <p></p> <div></div> rmsPrefix 匹配前缀 margin-left : marginLeft -webkit-margin-left : webkitMarginLeft -ms-margin-left : MsMarginLeft 和其他的不一样, M要大写 rdashAlpha 找到-和字符, 转大小写 -left : Left -2d : 2d</pre>	<pre>// Used for matching numbers core_pnum = /[+-]?(?:\d*\.)\d+(?:[eE][+-]?\d+)/.source, // Used for splitting on whitespace core_rnotwhite = /\S+/g, // A simple way to check for HTML strings // Prioritize #id over <tag> to avoid XSS via location.hash (#9521) // Strict HTML recognition (#11290: must start with <) rquickExpr = /^(?:\s*(<([\wW]+>)[^>]* #[([\w-]*)])\$/g, // Match a standalone tag rsingleTag = /^<(\w+)\s*\V?>(?:<\V1>)\$/g, // Matches dashed string for camelizing rmsPrefix = /^-ms-/g, rdashAlpha = /-([\da-z])/gi,</pre>
(85) 转驼峰的回调函数	<pre>// Used by jQuery.camelCase as callback to replace() fcamelCase = function(all, letter) { return letter.toUpperCase(); },</pre>
(90) DOM加载成功之后会触发的。819行相关。 先取消两个事件的绑定 , 然后调用工具方法jQuery.ready()。 最终只会走一次jQuery.ready() , 这个ready相关代码在382行。	<pre>// The ready event handler and self cleanup method completed = function() { document.removeEventListener("DOMContentLoaded", completed, false); window.removeEventListener("load", completed, false); jQuery.ready(); };</pre>

(96, 280) =====

(96, 280) 给jq对象添加方法和属性 , miaov7~12

jQuery.fn = jQuery.prototype = { //添加实例属性和方法

jquery : 版本
 constructor : 修正指向问题
 init() : 初始化和参数管理
 selector : 存储选择字符串
 length : this对象的长度
 toArray() : 转数组
 get() : 转原生集合
 pushStack() : JQ对象的入栈
 each() : 遍历集合
 ready() : DOM加载结构
 slice() : 集合的截取
 first() : 集合的第一项
 last() : 集合的最后一项
 eq() : 集合的指定项
 map() : 返回新集合
 end() : 返回集合前一个状态
 push() : (内部使用)
 sort() : (内部使用)

splice() : (内部使用)	
(98) core_version定义在49行 alert(\$.jquery); // 2.0.3	jquery.fn = jQuery.prototype = { // The current version of jQuery being used jquery: core_version,
(100) 修正属性，不懂的看下JS高程的面向对象编程。 要修正的原因是这里直接赋值了prototype，而不是添加函数	constructor: jQuery,
(101) context是前面的selector的执行上下文 \$('li','ul'); //ul下面的li	init: function(selector, context, rootjQuery) {

因为jQuery能够处理的类型很多，所以需要进行分配。在init中：

首先处理的是\$("")，\$(null)，\$(undefined)，\$(false)，这些都是不正确的元素

其次是html string (109, 175) 包括：\$('#div') \$('div') \$('#div1 div.box'); \$('') \$('12')

接下来DOMElement (176,180) 包括：\$(this) \$(document)

接下来是函数处理 (184,186) 包括：\$(function(){}) //文档加载

接下来处理数组和JSON形式 (188,193) 包括：\$([[]]) \$([{}])

(106) 错误的参数，直接返回	<pre> var match, elem; // HANDLE: \$(''), \$(null), \$(undefined), \$(false) if (!selector) { return this; } </pre>
(110) 判断是否是标签 if(){ \$('') \$('12') match1 = [null, '', null]; //\$('')的情况 match2 = [null, '12', null]; }else{ \$('#div1') \$('div') \$('#div1 div.box') \$('hello') //等同于\$('') rquickExpr 匹配的是 (标签+字符串) 或 (id) 的情况 exec找到匹配数组集，不仅匹配整体，还会匹配子项 正则后得到的结果 [完整字符串，含标签的项，含id的项] match3 = null ; //\$('div') \$('#div1 div.box') match4 = ['#div1', null, 'div1']; //\$('#div1') match5 = ['hello', '', null]; //\$('hello') }	<pre> // Handle HTML strings if (typeof selector === "string") { if (selector.charAt(0) === "<" && selector.charAt(selector.length - 1) === ">" && selector.length >= 3) { // Assume that strings that start and end with <> are HTML and skip the regex check match = [null, selector, null]; } else { match = rquickExpr.exec(selector); } } </pre>
(120) 能进入if(match && (match[1] !context))的情况： (match!=null, match[1]存在字符串含标签，context不存在 那么字符串是id，以上match1245都能进入)： \$(''), \$('#div1') 进一步判断， if(match[1]){ \$('') (123) }else{ \$('#div1') (150) }	<pre> // Match html or make sure no context is specified for #id if (match && (match[1] !context)) { // HANDLE: \$('html') -> \$(array) if (match[1]) { </pre>
(124) context 执行上下文，默认情况下是当前页面，还可以指定 iframe，那么创建的标签就是在iframe下，没有太多实际用途。 \$('', document) context instanceof jQuery为false， \$('', \$(document)) context instanceof jQuery为true 最后呢得到context = document	<pre> context = context instanceof jQuery ? context[0] : context; </pre>
(127) jQuery.parseHTML：字符串转节点数组 参数1：要转的字符串， 参数2：指定根节点，可以是document 参数3：true script标签能添加，false script标签不能添加 var str = "123<script> alert(1);</script>"; var arr = jQuery.parseHTML(str); //['li', 'li', 'li']; --- jQuery.merge：对外数组合并，对内部使用时，除了数组合并 还能够json合并 // var arr = ['a', 'b']; var arr = { 0: 'a', 1: 'b', length: 2 } var arr2 = ['c', 'd']; console.log(\$.merge(arr, arr2)); 按照上边的例子，执行完这步之后将得到一个jQuery对象 this = { 0: 'li', 1: 'li', 2: 'li', length: 3,其他jQuery方法等 }	<pre> // scripts is true for back-compat jQuery.merge(this, jQuery.parseHTML(match[1], context && context.nodeType ? context.ownerDocument context : document, true)); </pre>
(133) 针对下面这样一种叫的创建的情况 \$('', { title: 'hi', html: 'abcd',	<pre> // HANDLE: \$('html', props) </pre>

<pre>css:{ background : red})).appendTo('ul'); rsingleTag匹配单标签： 或 ，（不行： ） jQuery.isPlainObject：第二个参数必须是一个对象字面量，即 JSON 对传入的JSON进行for循环， 判断传入的是不是函数，这里的match可能是title或者html， \$.title()没有这个函数，\$.html() 有这个函数 有方法的时候对方法进行函数调用，this.html('abcd'); 没有就添加属性：this.attr(...) 创建标签走完了就需要return this 了，返回当前jQuery对象</pre>	<pre>if (rsingleTag.test(match[1]) && jQuery.isPlainObject(context)) { for (match in context) { // Properties of context are called as methods if possible if (jQuery.isFunction(this[match])) { this[match](context[match]); // ...and otherwise set as attributes } else { this.attr(match, context[match]); } } } return this;</pre>
<p>(149) 处理传值是id的情况</p> <p>match4 = ['#div1', null, 'div1']; //\$('#div1') 这种形式的， elem就是或得到的原生的document object</p> <p>---</p> <p>判断元素是否存在，直接判断elem在黑莓4.6下不靠谱，所以要 加elem.parentNode 更保险一些</p> <p>给length赋值，是因为前面存的是json的格式，没有length， 前面的使用了parseHTML是有length的，但是这里必须手动加</p> <p>处理下上下文和选择器的赋值。 返回这个对象 (这里可以大概这样理解：使用原生的找到这个对象，然后给 当前this jquery对象赋值，返回的是当前的this 即构造好后的 jquery对象)</p>	<pre>// HANDLE: \$(#id) } else { elem = document.getElementById(match[2]); // Check parentNode to catch when Blackberry 4.6 returns // nodes that are no longer in the document #6963 if (elem && elem.parentNode) { // Inject the element directly into the jQuery object this.length = 1; this[0] = elem; } this.context = document; this.selector = selector; return this; }</pre>
<p>(170) // HANDLE: \$(expr, \$(...))</p> <p>else if (执行上下文不存在 或 \$(document)这种情况) {</p> <p>rootjQuery = \$(document);</p> <p>返回 rootjQuery . find(selector);</p> <p>find是查找一些元素，可能是：</p> <p>\$(document).find('ul li.box'); find 会进一步调用sizzle</p> <p>)else(//存在上下文的情况</p> <p>比如传进去的是document或者\$(document)</p> <p>)</p> <p>例子：</p> <ul style="list-style-type: none">o \$('ul', document).find('li'); 走else得到： <p>jQuery(document).find();</p> <ul style="list-style-type: none">o \$('ul', \$(document)).find('li'); 走if(context.jquery)得 <p>到：jQuery(document).find();</p> <p>处理完以后最终结果就是jQuery(document).find();</p>	<pre>// HANDLE: \$(expr, \$(...)) } else if (!context context.jquery) { return (context rootjQuery).find(selector); // HANDLE: \$(expr, context) // (which is just equivalent to: \$(context).find(expr) } else { return this.constructor(context).find(selector); }</pre>
<p>(176) 处理选择的是节点的：\$(this), \$(document)</p> <p>先判断选择到的是否是节点，如果是节点，肯定有nodeType， 如果是节点，那就赋值，可以简单理解为将这个DOM节点构造 成jquery对象。</p> <p>节点的执行上下文就是它本身。</p>	<pre>// HANDLE: \$(DOMElement) } else if (selector.nodeType) { this.context = this[0] = selector; this.length = 1; return this; }</pre>
<p>(182) 处理函数的情况，主要是文档加载的情况。</p> <p>文档加载有三种方式</p> <p>\$(function()); //简写的方式</p> <p>\$(document).ready(function()); //最终的方式</p> <p>\$(document).on('ready', function());</p> <p>简写的方式通过右侧的函数处理，仍然调用的是最终的方式。</p>	<pre>// HANDLE: \$(function) // Shortcut for document ready } else if (jQuery.isFunction(selector)) { return rootjQuery.ready(selector); }</pre>
<p>(188) 处理\$('#div1') 这个情况</p> <p>\$('#div1') —处理后得到--> \$('#div1')</p>	<pre>if (selector.selector !== undefined) { this.selector = selector.selector; this.context = selector.context; }</pre>
<p>(193) 处理\$([]), \$([])这些个情况</p> <p>makeArray把类数组转成数组的一个方法。</p> <pre>var aDiv = document.getElementsByTagName('div'); \$.makeAarray(aDiv);</pre> <p>写两个参数，内部使用，转成了json，必须是特殊的json（包含length的）</p> <pre>var aDiv = document.getElementsByTagName('div'); console.log(\$.makeAarray(aDiv, {length:0}));</pre>	<pre>return jQuery.makeArray(selector, this); },</pre>
<p>(196) 存两个属性</p>	<pre>// Start with an empty selector selector: "", // The default length of a jQuery object is 0 length: 0,</pre>
<p>(202) 转数组</p> <pre>console.log(\$('div')); //得到json对象 console.log(\$('div').toArray()); //得到数组 \$('div') : {0:div, 1:div, 2:div, length:3} \$('div').toArray() : [div, div, div]</pre>	<pre>toArray: function() { return core_slice.call(this); },</pre>
<p>(208) 把jquery对象转成原生集合</p> <pre>\$('div').get(0).innerHTML = '222222'; --- for(var i=0 ; i< \$('div').get().length; i++){ \$('div').get(i).innerHTML = '222222'; } ---</pre>	<pre>// Get the Nth element in the matched element set OR // Get the whole matched element set as a clean array get: function(num) { return num == null ? // Return a 'clean' array this.toArray() : // Return just the object (num < 0 ? this[this.length + num] : this[num]); }</pre>

当num不存在时，转集合，调用的就是上面的toArray方法，转成数组， 如果num写了，然后判断下num值，在这个jquery的json中找到这个元素，然后返回。	}, (num < 0 ? this[this.length + num] : this[num]);			
(220) pushStack JQ对象的入栈处理 • \$('div').pushStack(\$('span')).css('background', 'red'); div先入栈，span后入栈，span变红 • this.constructor() 是一个空的jquery对象，因为elems可能是一个集合，所以需要merge。 • 然后呢，把div对象挂在到了 span对象的prevObject 属性下，执行上下文不变。 • 返回后入栈的对象 --- \$('div').pushStack(\$('span')).css('background', 'red').end().css('background', 'yellow'); //span变红，end回溯到栈的上一层，div变黄 --- \$('div').slice(1,3).css('background', 'red').end().css('color','blue'); 要明白，有一个栈： <table><tr><td></td></tr><tr><td>2.3div</td></tr><tr><td>4个div</td></tr></table> 第一个css操作的是2、3div，然后end回到最底层，第二个css操作的是4个div		2.3div	4个div	// Take an array of elements and push it onto the stack // (returning the new matched element set) pushStack: function(elems) { // Build a new jQuery matched element set var ret = jQuery.merge(this.constructor(), elems); // Add the old object onto the stack (as a reference) ret.prevObject = this; ret.context = this.context; // Return the newly-formed element set return ret; },
2.3div				
4个div				
(236) 这个each调用的是工具方法的each。 在jquery中，有很多即是工具方法又是实例方法的。工具方法是最底层的。	// Execute a callback for every element in the matched set. // (You can seed the arguments with an array of args, but this is // only used internally.) each: function(callback, args) { return jQuery.each(this, callback, args); },			
(240) 调用的是内部的工具方法。	ready: function(fn) { // Add the callback jQuery.ready.promise().done(fn); return this; },			
(247) 调用的是入栈的方法	slice: function() { return this.pushStack(core_slice.apply(this, arguments)); },			
(251) 找集合中的任意元素的方法。 \$('div').eq(0).css('background', 'red'); 通过入展，找到指定的元素添加进去，css操作之前，维持的栈是这样的： <table><tr><td></td></tr><tr><td>第一个div</td></tr><tr><td>4个div</td></tr></table>		第一个div	4个div	first: function() { return this.eq(0); }, last: function() { return this.eq(-1); }, eq: function(i) { var len = this.length, j = i + (i < 0 ? len : 0); return this.pushStack(j >= 0 && j < len ? [this[j]] : []); },
第一个div				
4个div				
(265) 调用工具方法中的map进行二次处理， var arr = ['a','b','c']; arr = \$.map(arr, function(elem,i){ return elem + i; }); alert(arr);	map: function(callback) { return this.pushStack(jQuery.map(this, function(elem, i) { return callback.call(elem, i, elem); })); },			
(271) 找到栈中下一层	end: function() { return this.prevObject this.constructor(null); },			
(277) 将数组的方法挂载到jQuery对象下面，jQuery对象就有了数组的方法	// For internal use only. // Behaves like an Array's method, not like a jQuery method. push: core_push, sort: [].sort, splice: [].splice };			

(285, 347) =====

(285, 347) extend : JQ的继承方法 miaov13-14

首先，你得明白继承的几种用法：

jQuery.extend这个叫做扩展静态方法，jQuery.fn.extend = jQuery.prototype.extend 这个是扩展实例方法。

\$.extend() Vs \$.fn.extend()

当只写一个对象自变量的时候，是JQ中扩展插件的形式。

\$.extend({ //这种叫扩展工具方法	\$.fn.extend({ //这种叫扩展JQ实例方法
aaa:function(){ alert(1); }, bbb:function(){ alert(2); } }); \$aaa(); //调用方式 \$.bbb();	aaa:function(){ alert(3); }, bbb:function(){ alert(4); } }); \$0.aaa(); //调用方式 \$0.bbb();

以上2个代可以放在一个页面执行

为什么可以用两种不同的方式呢？

\$.extend(); -> this 是\$ -> 那么添加this.aaa 就是添加\$.aaa

`$.fn.extend()`; -> 它的this是`$.fn`, 添加`this.aaa`, 就是在原型上添加aaa方法, 所以必须使用创建对象的方式来调用这个方法。所以呢, 可以用同一套代码(变量名方法名)来写出不同的功能

当写多个对象自变量时, 后面的对象都是扩展到第一个对象身上

```
var a = {};  
$.extend(a, {name: 'hello'}, {age: 30});  
console.log(a);  
--  
还可以做深拷贝和浅拷贝  
var a = {};  
var b = {name: 'hello', age: {old: 30}};  
$.extend(a, b); //深拷贝 $.extend(true, a, b);  
a.name = 'hi';  
a.age.old = 40;  
alert(b.name); //hello  
alert(b.age.old); //40, 如果b中的是一个对象的属性, 则会被修改
```

然后呢, 让我们看看这块代码的一个简化版本:

```
jQuery.extend = jQuery.fn.extend = function() {  
    定义一些变量  
    if() 看是不是深拷贝的情况  
    if() 看参数正确不正确  
    if() 看是不是插件情况  
    for() { 可能有多个对象的情况  
        if() 防止循环引用  
        if() 深拷贝  
        else if() 浅拷贝  
    }  
}
```

在JQ中: 拷贝继承

JS中: 类式继承 / 原型继承

(283) 看到61行源码分析处相关, 在jQuery.prototype下添加的任何方法, 可以通过jQuery.prototype.init new出来的对象使用	// Give the init function the jQuery prototype for later instantiation jQuery.fn.init.prototype = jQuery.fn;
(285) 定义了一些变量, target是目标元素, 因为后续的都是往第一个身上扩展, 所以target就是argument[0], 如果argument[0]不存在呢, 那就是一个新对象。 deep: 是否是深拷贝, 默认情况下不是深拷贝, 就是false	jQuery.extend = jQuery.fn.extend = function() { var options, name, src, copy, copyIsArray, clone, target = arguments[0] {}, i = 1, length = arguments.length, deep = false;
(293) 看是否是深拷贝 处理是否是深拷贝的情况, 因为如果是深拷贝的话, 那么第一个参数是boolean值, 那么target就是传入的第二个参数	// Handle a deep copy situation if (typeof target === "boolean") { deep = target; target = arguments[1] {}; // skip the boolean and the target i = 2; }
(301) 看参数是否正确 当你传入的目标元素不是一个对象, 或者不是一个函数的时候, 那就将其变成一个对象, 比如说当你传的是一个字符串啥的, 就会把它变成一个空对象。	// Handle case when target is a string or something (possible in deep copy) if (typeof target !== "object" && !jQuery.isFunction(target)) { target = {}; }
(306) 看是否是扩展插件的情况 因为i=1 (普通情况, 2深拷贝), 所以这里判断就是你传入的是不是一个元素的情况, 如果是一个元素呢, 那么就是在当前的jQuery对象上进行扩展, 目标元素就是this	// extend jQuery itself if only one argument is passed if (length === i) { target = this; --i; }
(311) 处理的是多个对象的情况 options存放的是后面的n多个对象, 然后判断其是否有值, 如果传递的是null的话, 是没有意义的。 src和copy存放的是两个变量, 最终的目的是要使得src = copy, 即要把options的对应属性扩展到target上	for (; i < length; i++) { // Only deal with non-null/undefined values if ((options = arguments[i]) != null) { // Extend the base object for (name in options) { src = target[name]; copy = options[name];
(320) 防止循环引用 什么是循环引用呢? 看下面例子: var a = {}; \$.extend(a, {name: a}); target 指的就是前面的a, copy指的就是后面的a	// Prevent never-ending loop if (target === copy) { continue; }
(325) 深拷贝 \$.extend(true, a, b); deep为真, 且copy是有值的, 并且b必须是一个对象, 或者b不是一个数组。 深拷贝利用的就是递归, 一层一层去找 定义空数组或者json的情况, 或者直接扩展原有的。 进一步递归。 var a = {name: {job: 'it'}}; var b = {name: {age: 30}}; \$.extend(true, a, b); 得到: a(name: { job:'it', age: 30});	// Recurse if we're merging plain objects or arrays if (deep && copy && (jQuery.isPlainObject(copy) (copyIsArray = jQuery.isArray(copy)))) { if (copyIsArray) { copyIsArray = false; clone = src && jQuery.isArray(src) ? src : []; } else { clone = src && jQuery.isPlainObject(src) ? src : {}; } // Never move original objects, clone them target[name] = jQuery.extend(deep, clone, copy);
(338) 浅拷贝 \$.extend(true, a, b); 如果b中的值是一个字符串或者数字的情况, 直接赋值	// Don't bring in undefined values } else if (copy !== undefined) { target[name] = copy; } }

```

    }
    // Return the modified object
    return target;
};

```


(349, 846) =====

(349, 846) jQuery.extend() : 扩展工具方法, 比如\$.trim();工具方法既可以给jQuery对象用, 也可以给原生js用, miaov15~28
 这部分的框架: 简化版本:

```

jQuery.extend({
  expando : 生成唯一JQ字符串 ( 内部 )
  noConflict() : 防止冲突
  isReady : DOM是否加载完 ( 内部 )
  readyWait : 等待多少文件的计数器 ( 内部 )
  holdReady() : 推迟DOM触发
  ready() : 准备DOM触发
  isFunction() : 是否为函数
  isArray() : 是否为数组
  isWindow() : 是否为window
  isNumeric() : 是否为数字
  type() : 判断数据类型
  isPlainObject() : 是否为对象自面量
  isEmptyObject() : 是否为空的对象
  error() : 抛出异常
  parseHTML() : 解析节点
  parseJSON() : 解析JSON
  parseXML() : 解析XML
  noop() : 空函数
  globalEval() : 全局解析JS
  camelCase() : 转驼峰
  nodeName() : 是否为指定节点名 ( 内部 )
  each() : 遍历集合
  trim() : 去前后空格
  makeArray() : 数组转真数组
  inArray() : 数组版indexOf
  merge() : 合并数组
  grep() : 过滤新数组
  map() : 映射新数组
  guid : 唯一标识符 ( 内部 )
  proxy() : 改this指向
  access() : 多功能值操作 ( 内部 )
  now() : 当前时间
  swap() : CSS交换 ( 内部 )
})

```

<p>(351) expando : jQuery203122394728178478 , 就是一个字符串,  expando和guid啥子区别呢?</p>	<pre> jQuery.extend({ // Unique for each copy of jQuery on the page expando: "jQuery" + (core_version + Math.random()).replace(/\D/g, ""), </pre>
<p>(353) 防冲突:</p> <p>例子:</p> <pre> var miaov = \$.noConflict(); var \$ = 123; miaov(function(){ alert(\$); }); //页面加载后会弹出123 </pre> <p>在41行有_\$ = window.\$, 这就是说, 即使你的\$ = 123是在引入jQuery库之前进行定义的, 会通过第一个if进行替换。</p> <pre> var \$ = 123; var jQuery = 456; <script src = "jquery-2.0.3.js"></script> var miaov = \$.noConflict(true); miaov(function(){ alert(\$); alert(jQuery); //不传true就还是构造函数, 传了true就是456了。 }); </pre> <p>所以第二个if就是专门解决当定义了变量jQuery的情况, 38行有:</p> <pre> jQuery = window.jQuery </pre>	<pre> noConflict: function(deep) { if (window.\$ === jQuery) { window.\$ = _\$; } if (deep && window.jQuery === jQuery) { window.jQuery = _jQuery; } return jQuery; }, </pre>

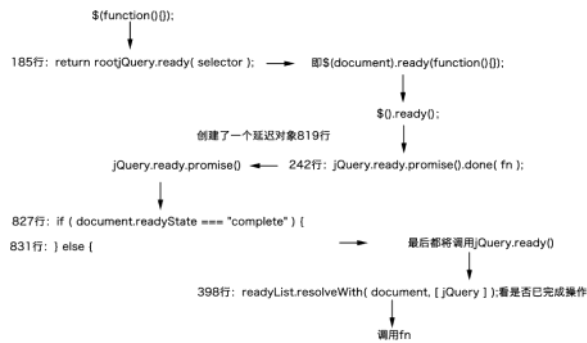
下面是一串关于dom加载的。首先补充点基础知识:

```

$(function(){ //等DOM加载完就会触发, 不用等图片啦、flash啦加载, 所以其加载速度快于window.onload
});
DOMMContentLoaded : DOM原生的加载完DOM触发的事件
window.onload = function(){ //等所有的加载完才会触发
};

```

具体的整个调用的流程可以看下图, 具体代码涉及 (819、90、382、373行等):



(366) 这几个参数用于对ready事件的延迟触发

```
$.holdReady(true); //推迟事件的触发
// $.holdReady(false); //释放事件的推迟，立即触发
$(function(){
    alert(123);
});
```

作用：
//a.js 内容：alert(1);
\$.holdReady(true);
\$.getScript('a.js', function(){
 \$.holdReady(false);
});
\$(function(){
 alert(2);
});

会出现弹出2先执行了，但是a.js还没有执行完，就会先弹出2后弹出1。
加上了holdReady就会先弹出1后弹出2。

源码的意思呢，就是只要hold为真，就进行++处理，为false就执行jQuery.ready()。hold的文件可能不只一个，调用几次，就++几回。然后如果传false，那么在385行会--，看这个变量是否为0了？

```
// Is the DOM ready to be used? Set to true once it occurs.
isReady: false,

// A counter to track how many items to wait for before
// the ready event fires. See #6781
readyWait: 1,

// Hold (or release) the ready event
holdReady: function( hold ) {
    if ( hold ) {
        jQuery.readyWait++;
    } else {
        jQuery.ready( true );
    }
},
```

(382) 处理dom.ready的

```
readyList.resolveWith( document, [ jQuery ] );
```

平时用的时候只使用readyList.resolve作为事件的触发，resolveWith是进行传参的处理，document是指向，[jQuery]就是参数，现在呢，document是jQuery.ready.promise().done(fn)中fn的this的指向，[jQuery]是fn的参数，

```
$(function(arg){
    alert(this); //document
    alert(arg); //jQuery
});
```

jQuery.fn.trigger和主动触发有关：

```
$(document).on('ready', function(){...});
```

先判断有没有主动触发的方法，如果有的话直接调用ready，调用完了之后再取消掉。

jQuery.isReady = true; 是让DOM只触发一次。

```
// Handle when the DOM is ready
ready: function( wait ) {

    // Abort if there are pending holds or we're already ready
    if ( wait === true ? --jQuery.readyWait : jQuery.isReady ) {
        return;
    }

    // Remember that the DOM is ready
    jQuery.isReady = true;

    // If a normal DOM Ready event fired, decrement, and wait if need be
    if ( wait !== true && --jQuery.readyWait > 0 ) {
        return;
    }

    // If there are functions bound, to execute
    readyList.resolveWith( document, [ jQuery ] );

    // Trigger any bound ready events
    if ( jQuery.fn.trigger ) {
        jQuery( document ).trigger("ready").off("ready");
    }
},
```

(409) 判断是否是函数

```
$.isFunction(show), type是判断不同对象的类型的
```

在IE的低版本下，typeof alert 为object。搜索那个bug，提供了一种可以判断所有的包括alert的方法。较复杂。

```
// See test/unit/core.js for details concerning isFunction.
// Since version 1.3, DOM methods and functions like alert
// aren't supported. They return false on IE (#2968).
isFunction: function( obj ) {
    return jQuery.type(obj) === "function";
},
```

(413) 使用的是原生的判断方法

```
alert(Array.isArray());
```

```
isArray: Array.isArray,
```

(415) 判断是不是window

```
alert($.isWindow(window));
```

obj!=null：不为null或者undefined，false==null (false)，判断原因是因为null和undefined是没有属性的，会报错。

```
isWindow: function( obj ) {
    return obj != null && obj === obj.window;
},
```

(419) 判断是不是数字，

```
typeof NaN //number
$.isNumeric(NaN) //false
```

首先判断是不是NaN，不是NaN再判断是不是有限的。

```
isNumeric: function( obj ) {
    return !isNaN( parseFloat(obj) ) && isFinite( obj );
},
```

(423) 判断数据类型，比原生的typeof强大很多。

```
var a = 'hello';
alert( $.type(a) );
```

在56行：core_toString = class2type.toString,
最靠谱的操作就是下面这种：

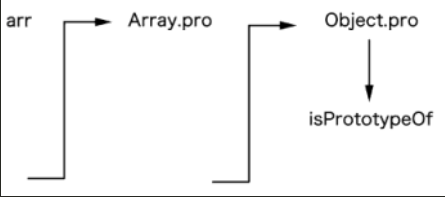
```
alert($.toString.call()); // [object Array]
alert($.toString.call(new Date)); // [object Date]
```

if(null || undefined)这两种类型，返回其字符串

如果传入的既不是object也不是function，那么就是基本类型，基本类型使用typeof就可以判断了。

在safari老版本中，使用typeof判断正则，会返回function，但是在新版本中，应该被返回object。

```
type: function( obj ) {
    if ( obj == null ) {
        return String( obj );
    }
    // Support: Safari <= 5.1 (functionish RegExp)
    return typeof obj === "object" || typeof obj === "function" ?
        class2type[ core_toString.call(obj) ] || "object" :
        typeof obj;
},
```


<p>class2type[core_toString.call(obj)] 这个，看到844行。</p> <p>(433) 判断是不是对象字面量，</p> <p>有两种形式会返回true：</p> <pre>//var obj = {name: 'hello'}; var obj = new Object(); // var obj = []; //数组，返回false alert(\$.isPlainObject(obj));</pre> <p>把一个DOM节点放到obj当中也会返回object，</p> <p>if(不是object的，DOM节点，window) 返回false</p> <p>---</p> <p>core_hasOwn.call(obj.constructor.prototype, "isPrototypeOf") 这个看到57行有：core_hasOwn = class2type.hasOwnProperty，</p>  <p>看上图要明白，arr的原型链上有Array.pro，Array.pro的原型链上有Object.pro，Object.pro的下面才有方法isPrototypeOf，所以如果不是Object，下面就没有hasOwn的方法isPrototypeOf</p> <p>火狐20以下，如果window.location执行次数比较多，会产生递归泄露的情况，所以要try一下。</p>	<pre>isPlainObject: function(obj) { // Not plain objects: // - Any object or value whose internal [[Class]] property is not "[object Object]" // - DOM nodes // - window if (jQuery.type(obj) !== "object" obj.nodeType jQuery.isWindow(obj)) { return false; } // Support: Firefox <20 // The try/catch suppresses exceptions thrown when attempting to access // the "constructor" property of certain host objects, ie. window.location // https://bugzilla.mozilla.org/show_bug.cgi?id=814622 try { if (obj.constructor && !core_hasOwn.call(obj.constructor.prototype, "isPrototypeOf")) { return false; } } catch (e) { return false; } // If the function hasn't returned already, we're confident that // obj is a plain object, created by {} or constructed with new Object return true; },</pre>
<p>(460) 判断是不是空对象</p> <pre>var obj = {}; //true var obj = {}; //true function Aaa(){} var obj = new Aaa; // true alert(\$.isEmptyObject(obj));</pre> <p>其实质就是通过forin实现，走了forin就返回false</p>	<pre>isEmptyObject: function(obj) { var name; for (name in obj) { return false; } return true; },</pre>
<p>(468) 抛出自定义的错误</p> <pre>\$.error("这是错误"); 出现错误的时候方便定位</pre>	<pre>error: function(msg) { throw new Error(msg); },</pre>
<p>(472) 解析节点</p> <pre>var str = ''; console.log(\$.parseHTML(str, document, true));</pre> <p>第一个参数：要转的字符串</p> <p>第二个参数：指定根节点</p> <p>第三个参数：script标签是否可用</p> <p>---</p> <p>首先判断第一个参数是不是字符串。</p> <p>因为可以省掉第二个参数，所以要判断第二个参数是不是boolean，如果是boolean，那么就把第二个值传给keepScript。</p> <p>再然后如果没有指定context，那么context就会被设置为document。</p> <p>判断是不是单标签。</p> <p>如果是单标签，就会创建这个单标签，然后返回。</p> <p>如果是多标签，那么先通过文档碎片的方式创建这个标签。</p> <p>注意，scripts标签是单独存在了scripts变量中。</p> <p>如果scripts是true，那么scripts标签就会被删除掉，如果scripts是false，那么scripts标签就不会被删掉。</p> <p>最后通过merge一下，将所有的转成数组返回。</p> <p>这部分 核心创建dom节点还是调用的jQuery.buildFragment</p>	<pre>// data: string of html // context (optional): If specified, the fragment will be created in this context, defaults to document // keepScripts (optional): If true, will include scripts passed in the html string parseHTML: function(data, context, keepScripts) { if (!data typeof data !== "string") { return null; } if (typeof context === "boolean") { keepScripts = context; context = false; } context = context document; var parsed = rsingleTag.exec(data), scripts = !keepScripts && []; // Single tag if (parsed) { return [context.createElement(parsed[1])]; } parsed = jQuery.buildFragment([data], context, scripts); if (scripts) { jQuery(scripts).remove(); } return jQuery.merge([], parsed.childNodes); },</pre>
<p>(502) JSON转换</p> <pre>var str = '{"name": "hello"}'; alert(\$.parseJSON(str).name); JSON.parse 是ECMA5支持的，IE8以上均支持。</pre>	<pre>parseJSON: JSON.parse,</pre>
<p>(505) 解析XML数据</p> <p>用法：</p> <pre>var xml = "<rss version='2.0'><channel><title>RSStitle</title></channel></rss>"; xmlDoc = \$.parseXML(xml), \$xml = \$(xmlDoc), \$title = \$xml.find("title"); ---</pre> <p>首先判断数据是否存在，不存在或者不是字符串类型就返回空</p> <p>new DOMParser() 创建了一个实例对象，然后用它下面的parseFromString方法即可。</p> <p>我们要解析的xml必须是完整的。</p> <p>在IE9下如果传入的是错误的xml，那么 tmp.parseFromString直接会报错，那么xml = undefined。</p> <p>在其他浏览器下，如果传入的错误的xml，那么 tmp.parseFromString 这个不会报错，但是会创建一个parsererror标签。</p>	<pre>// Cross-browser xml parsing parseXML: function(data) { var xml, tmp; if (!data typeof data !== "string") { return null; } // Support: IE9 try { tmp = new DOMParser(); xml = tmp.parseFromString(data , "text/xml"); } catch (e) { xml = undefined; } if (!xml xml.getElementsByTagName("parsererror").length) { jQuery.error("Invalid XML: " + data); } return xml; },</pre>
<p>(525) 返回一个空函数。</p> <p>写插件或组件的时候，会写一些默认参数。</p>	<pre>noop: function() {},</pre>
<p>(528) 全局解析JS</p> <pre>function(){} jQuery.globalEval(" var newVar = true;")</pre>	<pre>// Evaluates a script in a global context globalEval: function(code) { var script, indirect = eval;</pre>

<pre> } test(); alert(newVar); --- 首先进来先去一下空格。 然后进来判断下是否包含严格模式声明，如果包含，那么是不能用eval 的。 在if中，即严格模式下，通过创建script标签的形式，将其变成全局的。 在else中直接用的eval来解析字符串。 关于eval，解释上面为什么要先存eval： function test(){ var live = eval; live("var a=10;");//这里的live就是window.eval是一个全局的 属性 } test(); alert(a); //弹出10 --- function test(){ eval("var a=10;");//这里的eval是会被解析成关键字 } test(); alert(a); //错误，undefined a</pre>	<pre> indirect = eval, code = jQuery.trim(code); if (code) { // If the code includes a valid, prologue position // strict mode pragma, execute code by injecting a // script tag into the document. if (code.indexOf("use strict") === 1) { script = document.createElement("script"); script.text = code; document.head.appendChild(script).parentNode.removeChild(script); } else { // Otherwise, avoid the DOM node creation, insertion // and removal by using an indirect global eval indirect(code); } } }, }</pre>
<p>(552) 转驼峰的，将js中的样式转为大小写的方式，</p> <ul style="list-style-type: none">-ms-transform -> msTransform-moz-transform -> mozTransform <p>fcamelCase 是一个回调函数，将符合的子项转成大写</p>	<pre>// Convert dashed to camelCase; used by the css and data modules // Microsoft forgot to hump their vendor prefix (#9572) camelCase: function(string) { return string.replace(rmsPrefix, "ms-").replace(rdashAlpha, fcamelCase); },</pre>
<p>(556) 是否是指定的节点名，</p> <pre>alert(\$nodeName(document.documentElement, 'html')); //true</pre> <p>在不同的浏览器下获取到的名字可能是大写的也可能是小写的。</p>	<pre>nodeName: function(elem, name) { return elem.nodeName && elem.nodeName.toLowerCase() === name.toLowerCase(); },</pre>
<p>(561) each工具方法</p> <pre>var arr = ['a', 'b', 'c', 'd']; // json，getElementsByTagName \$.each(arr, function(i, value){ alert(i); });</pre> <p>isArraylike为真，数组/类数组操作，falsejson操作。this是true</p> <p>先判断是否是内部使用。</p> <p>如果是内部使用的话，和外部使用的不同之处就在于传了一个不定参数，然后就是使用call和apply的区别。</p>	<pre>// args is for internal usage only each: function(obj, callback, args) { var value, i = 0, length = obj.length, isArray = isArraylike(obj); if (args) { if (isArray) { for (; i < length; i++) { value = callback.apply(obj[i], args); if (value === false) { break; } } } else { for (i in obj) { value = callback.apply(obj[i], args); if (value === false) { break; } } } } } },</pre>
<p>(587) 不是内部使用的话else</p> <p>首先判断是不是数组类数组，如果是，那么就走for循环，如果不是，那就是json，那么就走for in 循环。</p> <pre>var json = { name : 'hello', age : 20}; \$.each(json, function(i, value){ alert(value); return false; //只会弹出hello，然后循环就返回了。在源码当中 就对应的是value===false，就退出了 });</pre>	<pre>// A special, fast, case for the most common use of each } else { if (isArray) { for (; i < length; i++) { value = callback.call(obj[i], i, obj[i]); if (value === false) { break; } } } else { for (i in obj) { value = callback.call(obj[i], i, obj[i]); if (value === false) { break; } } } } return obj; },</pre>
<p>(610) 去前后空格</p> <pre>var str = " 123 "; alert('(' + \$.trim(str) + ')');</pre> <p>58行有：core_trim = core_version.trim,这个是ECMA5中原生自带的一个方法。原生 str.trim() 。</p>	<pre>trim: function(text) { return text == null ? "" : core_trim.call(text); },</pre>
<p>(615) 类数组转真数组</p> <pre>var aDiv = document.getElementById("div"); console.log(\$.makeArray(aDiv)); var str = 'hello'; console.log(\$.makeArray(str));</pre> <p>写两个参数是对内使用，第二个参数json中必须带有length，而且属性名必须是012这种</p> <pre>console.log(\$.makeArray(str, {length: 0}));</pre> <p>想要走if里面的merge的话，传入的参数是要有长度的。</p> <p>isArraylike不能直接判断isArraylike(123)，所以需要转成对象。</p>	<pre>// results is for internal usage only makeArray: function(arr, results) { var ret = results []; if (arr != null) { if (isArraylike(Object(arr))) { jQuery.merge(ret, typeof arr === "string" ? [arr] : arr); } else { core_push.call(ret, arr); } } return ret; },</pre>
<p>(632) 数组版的indexOf</p> <pre>var arr = ['a', 'b', 'c', 'd']; alert(\$.inArray('b', arr)); //1</pre>	<pre>inArray: function(elem, arr, i) { return arr == null ? -1 : core_indexOf.call(arr, elem, i); },</pre>

<p>这里应该可以直接使用indexOf的吧？可以的</p> <pre>arr.indexOf('a') //0</pre>	
<p>(636) 对外使用：专门合并数组的。</p> <p>对内使用：转成特殊形式的json。</p> <p>首先获取到两个数组长度。</p> <p>如果第二个参数是json的话，那么很有可能没有长度，l就不是数字，走else，而且后一个json还必须是特殊的形式，即没有长度但是属性也必须是数字（012）的情况。</p> <p>走if的：</p> <pre>\$merge(['a', 'b'], ['c', 'd']);</pre> <pre>\$merge({ 0:'a', 1:'b', length:2 }, ['c', 'd']);</pre> <p>走else的：</p> <pre>\$merge(['a', 'b'], { 0:'c', 1:'d' });</pre> <p>循环完就设置length的值</p>	<pre>merge: function(first, second) { var l = second.length, i = first.length, j = 0; if (typeof l === "number") { for (; j < l; j++) { first[i++] = second[j]; } } else { while (second[j] !== undefined) { first[i++] = second[j++]; } } first.length = i; return first; },</pre>
<p>(656) 过滤得到新数组</p> <pre>var arr = [1,2,3,4]; arr = \$.grep(arr, function(value, i){ return value>2; }, true); //第3个参数有点像非，返回小于等于2的即1、2 console.log(arr); // [3,4] ,</pre> <p>既然是数组，那么肯定可以for循环，然后对每个元素进行循环调用，然后判断处理的结果。然后把返回的元素存到数组里。</p> <p>这货不是filter也是可以的么？</p> <pre>var arr = [1,2,3,4]; arr = arr.filter(function(value, i){ // a=[3,4] return value>2; });</pre>	<pre>grep: function(elems, callback, inv) { var retVal, ret = [], i = 0, length = elems.length; inv = !!inv; // Go through the array, only saving the items // that pass the validator function for (; i < length; i++) { retVal = !!callback(elems[i], i); if (inv !== retVal) { ret.push(elems[i]); } } return ret; },</pre>
<p>(676) 做映射</p> <pre>var arr = [1,2,3,4] arr = \$.map(arr, function(n){ return n+1; });</pre> <p>只要有length的就可以用for循环了，其他的就需要用for in循环了，所以要判断是不是类数组。</p> <p>然后开始计算结果，如果计算的结果不为空，那么就添到新数组里。</p> <p>这里没有直接返回ret的原因是，不想得到复合的数组，修改下源码返回ret：</p> <pre>arr = \$.map(arr, function(n){ return [n+1]; // [[1], [2], [3], [4]] });</pre>	<pre>// arg is for internal usage only map: function(elems, callback, arg) { var value, i = 0, length = elems.length, isArray = isArraylike(elems), ret = []; // Go through the array, translating each of the items to their if (isArray) { for (; i < length; i++) { value = callback(elems[i], i, arg); if (value !== null) { ret[ret.length] = value; } } // Go through every key on the object, } else { for (i in elems) { value = callback(elems[i], i, arg); if (value !== null) { ret[ret.length] = value; } } } // Flatten any nested arrays return core_concat.apply([], ret); },</pre>
<p>(709) 唯一标识符，他的作用和jquery中的事件操作非常相关</p>	<pre>// A global GUID counter for objects guid: 1,</pre>
<p>(713) 修改this指向</p> <pre>function show(n1, n2){ alert(this); } show(); // window \$.proxy(show, document)(3, 4); //document \$.proxy(show, document, 3, 4)(); //document \$.proxy(show, document, 3)(4); //document</pre> <p>传参这么复杂的原因是因为前面那部分的传参可以使在不运行show的情况下传参的。</p> <p>如果指向是一个字符串，那么做一个处理，像下面这样的情况：</p> <pre>var obj = { show: function(){ alert(this); } }; \$(document).click(obj.show); //document</pre> <p>现在呢，我想让这个this指向obj：</p> <pre>\$(document).click(\$.proxy(obj.show, obj)); //Object</pre> <p>或者这么写：</p> <pre>\$(document).click(\$.proxy(obj, 'show')); //Object，这样写也会转成上面那一种</pre> <p>因为this必须得是函数下面的才行，所以如果fn不是函数，那么就是undefined了。</p> <p>合并index>=2的参数</p> <p>arguments 不是数组，所以需要通过原生的转成真正的数组，然后再concat。</p> <p>然后呢，设置唯一标识。</p>	<pre>// Bind a function to a context, optionally partially applying any // arguments. proxy: function(fn, context) { var tmp, args, proxy; if (typeof context === "string") { tmp = fn[context]; context = fn; fn = tmp; } // Quick check to determine if target is callable, in the spec // this throws a TypeError, but we will just return undefined. if (!jQuery.isFunction(fn)) { return undefined; } // Simulated bind args = core_slice.call(arguments, 2); proxy = function() { return fn.apply(context this, args.concat(core_slice.call(arguments))); }; // Set the guid of unique handler to the same of original handler, so it can be removed proxy.guid = fn.guid = fn.guid jQuery.guid++; return proxy; },</pre>
<p>(742) 主要是内部使用的，多功能值的操作，</p>	<pre>// Multifunctional method to get and set values of a collection</pre>

<p>什么是多功能值的操作？</p> <pre><code>\$.css(), \$.attr(); set/get, 通过这一个方法可以get/set</code></pre> <pre><code>\$(function(){ alert(\$('#div').css('background')); \$('#div').css('background', 'yellow'); \$('#div').css({ background: 'yellow', width: '300px' }); });</code></pre> <p>参数：操作的元素或集合，回调函数，key指的是background/width，value就是要设置的值，chainable为真-设置值假-获取值，key有值的情况下bulk是false，设置多组值，一个json，如果是object的话，那么就是要设置json中的值，那么就会将chainable变量手动变为true（因为可以找到6k多行css调用这个函数的时候是判断参数个数的，如果是一个参数，传进来的值为假），然后会不断用递归设置值</p>	<pre><code>// Access functions: method to get and set values on a collection // The value/s can optionally be executed if it's a function access: function(elems, fn, key, value, chainable, emptyGet, raw) { var i = 0, length = elems.length, bulk = key == null; // Sets many values if (jQuery.type(key) === "object") { chainable = true; for (i in key) { jQuery.access(elems, fn, i, key[i], true, emptyGet, raw); } } // Sets one value } else if (value !== undefined) { chainable = true; if (!jQuery.isFunction(value)) { raw = true; } if (bulk) { // Bulk operations run against the entire set if (raw) { fn.call(elems, value); fn = null; } // ...except when executing function values } else { bulk = fn; fn = function(elem, key, value) { return bulk.call(jQuery(elem), value); }; } if (fn) { for (; i < length; i++) { fn(elems[i], key, raw ? value : value.call(elems[i], i, fn(elems[i], key))); } } } return chainable ? elems : // Gets bulk ? fn.call(elems) : length ? fn(elems[0], key) : emptyGet; },</code></pre>
<p>(795) 设置一个值的情况。</p> <p>这种情况下value一定要有值，然后判断下value是不是函数，在value是字符串情况下raw为真。</p> <p>在没有key值的情况下bulk为真，此时操作的是回调，if(bulk)里面主要针对value是函数的情况，进行一些设置。</p> <p>如果传的value不是函数，那么if(fn)里面就走fn(elems[i], key, value)</p> <p>如果value是函数，那么就走：</p> <pre><code>fn(elems[i], key, value.call(elems[i], i, fn(elems[i], key)))</code></pre> <p>前面一直都是设置。获取是在return这，没有key值会触发回调，有key值判断length存在不？然后返回集合中的值，如果没有length，那么就会返回undefined的值。</p> <p>?????绕死了。。</p>	<pre><code>// A method for quickly swapping in/out CSS properties to get correct calculations. // Note: this method belongs to the css module but it's needed here for the support module. // If support gets modularized, this method should be moved back to the css module. swap: function(elem, options, callback, args) { var ret, name, old = {}; // Remember the old values, and insert the new ones for (name in options) { old[name] = elem.style[name]; elem.style[name] = options[name]; } ret = callback.apply(elem, args []); // Revert the old values for (name in options) { elem.style[name] = old[name]; } return ret; };</code></pre>
<p>(793) 获取当前时间的</p> <pre><code>\$.now</code> //一串数字距离1970年1月1日的毫秒数</pre>	<pre><code>now: Date.now,</code></pre>
<p>(798) CSS交换的一个方法，内部使用。</p> <p>jq中获得隐藏元素的值display:none，会用到这个swap。</p> <pre><code>alert(\$('#div').width()); alert(\$('#div').get(0).offsetWidth);</code></pre> <p>如果想得到和display:none一样的效果，那么就需要：</p> <pre><code>display:block; visibility:hidden; position:absolute;</code></pre> <p>先把样式存到old，然后获取到css样式后，然后再变回来。</p>	<pre><code>jQuery.ready.promise = function(obj) { if (!readyList) { readyList = jQuery.Deferred(); // Catch cases where \$(document).ready() is called after the browser event has already occurred. // we once tried to use readyState "interactive" here, but it caused issues like the one // discovered by ChrisS here: http://bugs.jquery.com/ticket/12282#comment:15 if (document.readyState === "complete") { // Handle it asynchronously to allow scripts the opportunity to delay ready setTimeout(jQuery.ready); } else { // Use the handy event callback document.addEventListener("DOMContentLoaded", completed, false); // A fallback to window.onload, that will always work window.addEventListener("load", completed, false); } } return readyList.promise(obj); };</code></pre>
<p>(819) jQuery.ready.promise</p> <p>readyList 第一次进来的时候是个空对象，是没有的，只要一次加载成功，后续的都可以触发，所以只需要走一次就行了。</p> <p>首先创建的是一个延迟对象。</p> <p>DOM已经加载好的标志就是document.readyState === "complete"，加载完成的话，就直接调用工具方法即可。</p> <p>加载定时器是针对IE的，让它延后触发。</p> <p>else是正常情况，dom没有加载完，需要进行监测，监测两个时间的原因是，火狐在有缓存的情况下会先触发load事件，后触发DOMContentLoaded事件。为了最快加载，所以最好两个都写。不管走哪一个，都会调用completed这个回调。</p> <p>这个回调在第90行。</p> <p>return readyList.promise(obj)；延迟对象的状态有完成和未完成等，这些状态是可以被修改的，但是promise有一个特点，就是不想让这个状态被修改。</p>	<pre><code>// Populate the class2type map jQuery.each("Boolean Number String Function Array Date RegExp Object Error".split(" "), function(i, name) { class2type["[object " + name + "]"] = name.toLowerCase(); });</code></pre>
<p>(844) 遍历下各种类型，然后得到各种类型的纯小写，然后再存起来方便这里使用。</p>	<pre><code>function isArraylike(obj) { var length = obj.length, type = jQuery.type(obj);</code></pre>
<p>(848) 判断是不是数组或类数组。</p> <p>首先判断下window，因为下面判断时要判断一下属性，难免window下</p>	

会挂载一些length属性。
判断一下是不是一组元素节点。
然后判断下是不是数组，如果是数组，那就是真。
然后再判断是不是函数，因为函数下有可能也挂载了length属性。
最后（）里面的一长串判断的就是arguments：
function show(){
 \$.isArraylike(arguments);
}
show('a','b','c');
???其实这里没法判断(length-1)是不是存在obj中的？为什么要判断？？

```
if ( jQuery.isWindow( obj ) ) {  
    return false;  
}  
  
if ( obj.nodeType === 1 && length ) {  
    return true;  
}  
  
return type === "array" || type !== "function" &&  
    ( length === 0 ||  
      typeof length === "number" && length > 0 && ( length - 1 ) in obj );  
}
```

(877, 2856) =====

(2880, 3042) =====

Callbacks：回调对象：作用是函数的统一管理，miaov29-32

基本使用：

```
function aaa(){  
    alert(1);  
}  
function bbb(){  
    alert(2);  
}  
var cb = $.Callbacks();  
cb.add( aaa );  
cb.add( bbb ); //优点类似于事件绑定，添加多少个 都会执行。 这个就是观察者模式。
```

cb.fire(); //弹出1,2

对不同作用域下的函数做同一的管理

```
function aaa(){ alert(1); }  
(function(){  
    function bbb(){ alert(2); }  
})();  
aaa();  
bbb(); //只会弹出1，不会弹出2，因为bbb是局部的
```

```
var cb = $.Callbacks();  
function aaa(){ alert(1); }  
cb.add(aaa);  
(function(){  
    function bbb(){ alert(2); }  
    cb.add(bbb);  
})();  
cb.fire(); //弹出1,2
```

参数的使用：

```
var cb = $.Callbacks();  
// var cb = $.Callbacks('once'); fire只能触发一次  
cb.add(aaa);  
cb.add(bbb);  
cb.fire(); //弹出1,2  
cb.fire(); //不加参数，会第二次弹出1,2
```

```
var cb = $.Callbacks('memory'); fire只能触发一次  
cb.add(aaa);  
cb.fire(); //不加参数只弹出1，加了参数弹出1,2  
cb.add(bbb);
```

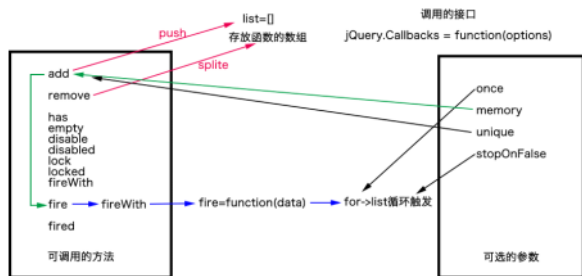
```
var cb = $.Callbacks('unique'); 去重，重复的函数名不被添加  
cb.add(aaa);  
cb.add(aaa);  
cb.fire(); //不加参数弹出1，1，加了参数弹出1,
```

```
function aaa(){ alert(1); return false; }  
var cb = $.Callbacks('stopOnFalse'); 添加的函数在执行过程中如果返回了false，后续函数不会被执行  
cb.add(aaa);  
cb.add(bbb);  
cb.fire(); //不加参数弹出1，2，加了参数弹出1,
```

--- 组合形式

```
$.Callbacks('once memory');
```

整个的设计思想图：



<p>(2846)</p> <p>core_rnotwhite对空格进行分割</p> <p>如果传值'once memory' :</p> <pre>optionsCache = { 'once memory' : {once:true, memory:true} }</pre> <p>options = {once:true, memory:true}; //返回的object就是options</p>	<pre>// String to Object options format cache var optionsCache = {}; // Convert String-formatted options into Object-formatted ones and store in cache function createOptions(options) { var object = optionsCache[options] = {}; jQuery.each(options.match(core_rnotwhite) [], function(_, flag) { object[flag] = true; }); return object; }</pre>
	<pre>/* * Create a callback list using the following parameters: * * options: an optional list of space-separated options that will change how * the callback list behaves or a more traditional option object * * By default a callback list will act like an event callback list and can be * "fired" multiple times. * * Possible options: * * once: will ensure the callback list can only be fired once (like a Deferred) * * 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) * * unique: will ensure a callback can only be added once (no duplicate in the list) * * stopOnFalse: interrupt callings when a callback returns false */</pre>
<p>(2880) 回调对象函数的开始</p> <p>options值的处理, 有单值情况, 多值情况, 无值情况</p> <p>如果传的值是"once memory"</p> <p>那么options = {once:true, memory:true};</p>	<pre>jQuery.Callbacks = function(options) { // Convert options from String-formatted to Object-formatted if needed // (we check in cache first) options = typeof options === "string" ? (optionsCache[options] createOptions(options)) : options; jQuery.extend({}, options);</pre>
<p>(2888) 定义一堆变量</p> <p>memory : true-表示add的时候会立即执行, false, add的时候不会执行</p> <p>fired : 执行过fire函数了就会标记为true, 没有执行则为false</p> <p>firing : true表示正在触发的过程中, false表示一次触发已经完成</p> <p>firingStart :</p> <p>firingLength :</p> <p>firingIndex :</p> <p>list : 所有的回调函数对象添加到那个列表, 图中的</p> <p>stack : 当写了once时, 为false, 否则为真</p>	<pre>var // Last fire value (for non-forgettable lists) memory, // Flag to know if list was already fired fired, // Flag to know if list is currently firing firing, // First callback to fire (used internally by add and fireWith) firingStart, // End of the loop when firing firingLength, // Index of currently firing callback (modified by remove if needed) firingIndex, // Actual callback list list = [], // Stack of fire calls for repeatable lists stack = !options.once && [],</pre>
<p>(2905) fire函数, 执行时, 最终都调用的这个</p> <p>里面是对list做一个for循环,</p> <p>只要一进来, fired就会存为true, 表示已经触发过一次了,</p> <p>然后保存索引, 起始值和长度,</p> <p>firing表示正在触发当中,</p> <p>然后循环触发,</p> <p>firing = false; 表示触发过程结束了。</p> <p>在循环过程中呢, 就会把参数data传给list中已经添加的每一个函数, 如果你的函数中有return false, 并且写了参数stopOnFalse, 那就会停下来, break掉, 并且在接下来的过程中, memory也会失效。</p> <p>if(stack)这一块的作用, 处理下面这种情况 :</p> <pre>function aaa(){ alert(1); cb.fire(); } function bbb(){ alert(2); } var cb = \$.Callbacks(); cb.add(aaa, bbb); cb.fire(); //会进入死循环, 一直弹出1,2,1,2,1,2...</pre> <p>stack是保证按顺序执行, 不会出现弹出11111的情况,</p> <p>如果传的参数中没有once的话, stack会是一个数组, 添加值操作在3023行, 就是把要执行的函数放进去, 所以这里能保证顺序执行</p> <p>如果参数是"once"的话stack为false, 如果有memory参数, 会清空list, 如果没有, 会走else, disable会阻止后续的任何操作。</p> <p>如果有"memory", 会清空数组, 第二次调用fire的时候就执行的是空数组了。</p>	<pre>// Fire callbacks fire = function(data) { memory = options.memory && data; fired = true; firingIndex = firingStart 0; firingStart = 0; firingLength = list.length; firing = true; for (; list && firingIndex < firingLength; firingIndex++) { if (list[firingIndex].apply(data[0], data[1]) === false && options.stopOnFalse) { memory = false; // To prevent further calls using add break; } } firing = false; if (list) { if (stack) { if (stack.length) { fire(stack.shift()); } } else if (memory) { list = []; } else { self.disable(); } } },</pre>

<p>(2932) self最后会被return，他就是Callback</p> <p>(2934) add</p> <p>首先进来判断list里面是否已存在，空也是存在的。所以总会走if。匿名函数自执行，针对这样一种情况：cb.add(aaa, bbb);这里的args就是[aaa, bbb]，然后对它进行一个foreach循环，如果传递的参数是一个函数，然后判断是否有unique，如果没有unique是可以push的，如果有unique，就要判断当前函数是否已经添加到数组中，如果没有，则push。</p> <p>不是函数的情况：</p> <pre>cb.add([aaa, bbb]); elseif就针对这种不是函数的情况。</pre> <p>2947行的add(arg)调用的是什么？</p> <p>如果当前正在执行一个队列中的函数，那么更新下执行长度firingLength。</p> <p>如果firing = false，当前没有函数执行，但是有memory参数，那么立即执行这个新添加到队列的函数。</p> <p>memory变量在上面有定义，第一次的时候是undefined，所以是不会走elseif的，fire (2905) 一次后，如果有memory这个参数，memory变量会为true，然后呢，这个地方elseif就会为真，然后就会再次调用fire。所以可以这么理解memory参数，第一次调用fire时，其内还是执行的aaa，但是添加cb.add(bbb)时，因为有memory参数，并且已执行过cb.fire，所以在cb.add(bbb)时会同时执行bbb。</p>	<pre>// Actual Callbacks object self = { // Add a callback or a collection of callbacks to the list add: function() { if (list) { // First, we save the current length var start = list.length; (function add(args) { jQuery.each(args, function(_, arg) { var type = jQuery.type(arg); if (type === "function") { if (!options.unique !self.has(arg)) { list.push(arg); } } else if (arg && arg.length && type !== "string") { // Inspect recursively add(arg); } }); })(arguments); // Do we need to add the callbacks to the // current firing batch? if (firing) { firingLength = list.length; // With memory, if we're not firing then // we should call right away } else if (memory) { firingStart = start; fire(memory); } } return this; }, }</pre>
<p>(2965) remove 也可以处理多个参数</p>	<pre>// Remove a callback from the list remove: function() { if (list) { jQuery.each(arguments, function(_, arg) { var index; while((index = jQuery.inArray(arg, list, index)) > -1) { list.splice(index, 1); // Handle firing indexes if (firing) { if (index <= firingLength) { firingLength--; } if (index <= firingIndex) { firingIndex--; } } } }); } return this; },</pre>
<p>(2987) has</p> <p>不传参数的时候就会判断list中是否还有需要执行的函数</p>	<pre>// Check if a given callback is in the list. // If no argument is given, return whether or not list has callbacks attached. has: function(fn) { return fn ? jQuery.inArray(fn, list) > -1 : !(list && list.length); },</pre>
<p>(2991) empty 情况整个数组，所有的return this，都是用于可以链式操作的。</p> <p>(2997) disable 后面的全部都禁止</p> <p>(3002) disabled 判断当前是否是禁止的</p> <p>(3006) lock：</p> <pre>var cb = \$.Callbacks('memory'); cb.add(aaa); cb.fire(); //cb.disable(); // 执行后只会弹出1 cb.lock(); //把后续的fire锁住，会弹出1,2 cb.add(bbb);</pre> <p>(3014) locked：</p> <p>判断锁没锁住</p>	<pre>// Remove all callbacks from the list empty: function() { list = []; firingLength = 0; return this; }, // Have the list do nothing anymore disable: function() { list = stack = memory = undefined; return this; }, // Is it disabled? disabled: function() { return !list; }, // Lock the list in its current state lock: function() { stack = undefined; if (!memory) { self.disable(); } return this; }, // Is it locked? locked: function() { return !stack; },</pre>
<p>(3018) fireWith</p> <p>首先判断list是否存在，第一次的时候fired是undefined，所以走if。fire中参数可以作为回调的参数的：</p> <pre>function aaa(n){ alert('aaa' + n); } function bbb(n){ alert('bbb' + n); } var cb = \$.Callbacks(); cb.add(aaa, bbb); cb.fire('hello'); //会弹出aaahello和bbbhello</pre> <p>这里的参数传递就是args。</p> <p>如果已经执行过一次了，那么fired就是true，那如果想走if，就要看stack了，stack定义在2903行，stack = options.once && [], 如果有once，那么stack就是false，否则是[]。</p>	<pre>// Call all callbacks with the given context and arguments fireWith: function(context, args) { if (list && (!fired stack)) { args = args []; args = [context, args.slice ? args.slice() : args]; if (firing) { stack.push(args); } else { fire(args); } } return this; },</pre>
<p>(3031) fire调用的是fireWwith，把当前的对象和参数传过去</p>	

<p>❓ 这个地方如此简单，那为什么不直接去掉fireWith呢？直接用fire不就好了，包裹那么多层做什么？</p>	<pre>// Call all the callbacks with the given arguments fire: function() { self.fireWith(this, arguments); return this; },</pre>
<p>(3036) fired 判断当前有没有触发过</p>	<pre>// To know if the callbacks have already been called at least once fired: function() { return !!fired; } }; return self; };</pre>

(3043, 3183) =====

(3043, 3183) Deferred：延迟对象：作用是对异步的统一管理，miaov33~38

首先来看这部分框架：

```
jQuery.extend({
    Deferred: function() {},
    when: function() {}
});
```

\$.Deferred(); 延迟对象是基于 \$.Callbacks(); 开发的。

\$.when();

用法：

<pre>var cb = \$.Callbacks(); setTimeout(function(){ alert(111); cb.fire(); }, 1000); cb.add(function(){ alert(222); });</pre> <p>运行结果：先弹111，后弹222</p>	<pre>var dfd = \$.Deferred(); setTimeout(function(){ alert(111); dfd.resolve(); }, 1000); dfd.done(function(){ alert(222); });</pre> <p>运行结果：先弹111，后弹222</p>
--	--

所以呢，fire 类似于 resolve，add 类似于 done

对异步的统一管理：

```
setTimeout(function(){
    alert(111);
}, 1000);
alert(222);
```

在延迟对象中还可以用很多其他的方法：

```
var dfd = $.Deferred();
setTimeout(function(){
    alert(111);
    dfd.reject();
}, 1000);
dfd.fail(function(){
    alert(222);
});
```

一共有三套组合：

notify progress
resolve done
reject fail

区别：

- 1.resolve 和 reject 只会触发一次，notify 可以触发多次
- 2.如果 resolve 已经调用完成，那么再进行 done 操作的时候，里面的东西都会立即触发。

```
var dfd = $.Deferred();
setTimeout(function(){
    alert(111);
    dfd.resolve();
}, 1000);
dfd.done(function(){
    alert('aaa');
});
$('.input').click(function(){
    dfd.done(function(){
        alert('bbb'); // 谈完111，aaa 后 点击后会立即触发。
    });
});
```

当你了解了延迟对象之后，你可以这么去写：

```
$.ajax({
    url: 'xxx.php',
    success: function(){
        alert('成功');
    },
    error: function(){
        alert('失败');
    }
});
```



```

    }
  });
  $.ajax('xxx.php').done(function(){ alert('成功'); }).fail( function(){ alert('失败'); });

```

里面对象promise和deferred区别：

promise	deferred
state	resolve
always	reject
then	notify
promise	
pipe	state
done	always
fail	then
progress	promise
	pipe
	done
	fail
	progress

然后呢，多了几个resolve reject和notify有什么用呢？

```

function aaa(){
  var dfd = $.Deferred();
  setTimeout(function(){
    dfd.resolve();
  }, 1000);
  return dfd; //如果不想让状态被修改掉，return dfd.promise(); 并且修改状态会被报错。
}

var newDfd = aaa(); //这里一定要存下来，如果不存下来，下面就是两个独立的aaa了
newDfd.done(function(){
  alert('成功');
}).fail(function(){
  alert('失败');
});

newDfd.reject(); //只会弹出失败，因为失败会比成功先触发，触发完后就不会走成功了。说明状态很容易被修改掉。

```

(3043) 这里一上来就定义了一个数组，是表示对应关系的，第三个参数就是对应的回调函数，最后一个参数呢，是一个状态。
有了这样一个映射之后呢，前面三个resolve, reject, notify调用的都是fire这个函数，接下来三个呢，done,fail,progress调用的是回调对象中的add，那么他们怎么建立映射关系呢？通过这个数组，还有3095的each方法。
传递once memory区别：

```

setInterval(function(){
  alert(111);
  dfd.resolve();
  // dfd.reject()
},1000);
dfd.done(function(){
  alert('成功');
}).fail(function(){
  alert('失败');
}).progress(function(){
  alert('进行中');
});

```

```

jQuery.extend({
  Deferred: function( func ) {
    var tuples = [
      // action, add listener, listener list, final state
      [ "resolve", "done", jQuery.Callbacks("once memory"), "resolved" ],
      [ "reject", "fail", jQuery.Callbacks("once memory"), "rejected" ],
      [ "notify", "progress", jQuery.Callbacks("memory") ]
    ],

```

(3052) 这是一个状态，完成/未完成，可能出现的值是：pending，resolved，rejected，它会在遍历的时候被修改成tuple[3]，代码3097

```

state = "pending",

```

(3057) always：总是触发，
var dfd = \$.Deferred();
setTimeout(function(){
 dfd.reject();
 //dfd.resolve();
}, 1000);
dfd.always(function(){
 alert('hello'); //不管已完成还是未完成都会走hello
});
(3061) then：是一种简写的方式
dfd.then(function(){
 alert('成功的');
}, function(){
 alert('失败的');
}, function(){
 alert('正在进度中的');
});
遍历每一个参数。fn 这里期待得到函数，不是函数就是false。
deferred[tuple[1]](function() {，对应的函数走对应的回调，
var returned...判断函数有没有，再执行函数，然后argument就是可以传递的参数，比如：
dfd.reject('hi');
dfd.then(function(){
 alert('成功的');
}, function(){
 alert(arguments[0]);
}, function(){
 alert('正在进度中的');
});
returned 和下面的if判断，是针对pipe的。

```

promise = {
  state: function() {
    return state;
  },
  always: function() {
    deferred.done( arguments ).fail( arguments );
    return this;
  },
  then: function( /* fnDone, fnFail, fnProgress */ ) {
    var fns = arguments;
    return jQuery.Deferred(function( newDefer ) {
      jQuery.each( tuples, function( i, tuple ) {
        var action = tuple[ 0 ],
            fn = jQuery.isFunction( fns[ i ] ) && fns[ i ];
        // deferred[ done | fail | progress ] for forwarding actions to newDefer
        deferred[ tuple[1] ](function() {
          var returned = fn && fn.apply( this, arguments );
          if ( returned && jQuery.isFunction( returned.promise() ) ) {
            returned.promise()
              .done( newDefer.resolve )
              .fail( newDefer.reject )
              .progress( newDefer.notify );
          } else {
            newDefer[ action + "With" ]( this === promise ?
              newDefer.promise() : this, fn ? [ returned ] : arguments );
          }
        });
      });
      fns = null;
    }).promise();
  },

```

<pre>var dfd = \$.Deferred(); setTimeout(function(){ dfd.resolve('hi'); }, 1000); var newDfd = dfd.pipe(function(){ return arguments[0] + 'miaov'; }); newDfd.done(function(){ alert(arguments[0]); //弹出himiaov });</pre>	
<p>(3085) promise , 从3121行过来执行这个方法 , obj是有参数的是deferred , 那么会走jQuery.extend(obj, promise) , 那就是说把promise下面的所有方法都继承给deferred。</p> <p>如果不带参数呢 , 那么就会直接返回promise对象 , 这个对象是不带状态的 , 那么在外边就不能修改它的状态。</p>	<pre>// Get a promise for this deferred // If obj is provided, the promise aspect is added to the object promise: function(obj) { return obj != null ? jQuery.extend(obj, promise) : promise; },</pre>
<p>(3089) deferred对象的定义</p>	<pre>deferred = {};</pre>
<p>(3092) then和pipe是一个方法 , 功能是不一样的 , 但是是用同一套代码写出来的。</p>	<pre>// Keep pipe for back-compat promise.pipe = promise.then;</pre>
<p>(3095) 对映射数组进行遍历操作。 数组第二项 , 即done , fail , progress , 对应回调对象的add的 , 得到的list里面存放的是回调对象。 下面的stateString就是状态 , 再然后呢 , 就把promise[done fail progress] = list.add , 这个就很清楚了 , done fail progress就是add ,</p> <p>(3103) 只有resolve和reject才会走if (stateString) , 如果是notify , 是没有状态的。 在这个if中 , 又添加了一个方法 , 将state状态修改 , 然后下面的处理是这样的 , 一旦触发了完成 , 就不会再去触发未完成 , 一旦触发了未完成 , 就不会再去触发完成。这个就是通过tuples[i ^ 1][2].disable , tuples[2][2].lock 来实现的 , tuples[i ^ 1][2].disable -> 如果我先走的done , 那么就对tuples[0 ^ 1][2].disable -> 即对fail进行disable</p> <p>(3113) 状态调用的就是fireWith方法 : deferred也是延迟对象。</p>	<pre>// Add list-specific methods jQuery.each(tuples, function(i, tuple) { var list = tuple[2], stateString = tuple[3]; // promise[done fail progress] = list.add promise[tuple[1]] = list.add; // Handle state if (stateString) { list.add(function() { // state = [resolved rejected] state = stateString; // [reject_list resolve_list].disable; progress_list.lock }, tuples[i ^ 1][2].disable, tuples[2][2].lock); } // deferred[resolve reject notify] deferred[tuple[0]] = function() { deferred[tuple[0] + "With"](this === deferred ? promise : this, arguments); }; deferred[tuple[0] + "With"] = list.fireWith; });</pre>
<p>(3121) 这里执行会到3085</p>	<pre>// Make the deferred a promise promise.promise(deferred);</pre>
	<pre>// Call given func if any if (func) { func.call(deferred, deferred); } // All done! return deferred; },</pre>
<p>(3133) when 主要针对多延迟对象的操作</p> <pre>var dfd = \$.Deferred(); dfd.done(); --- \$.when().done(); --- \$.when() 和dfd区别 : dfd只能对一个延迟对象做操作 , \$.when()可以对多个延迟对象做操作。 --- aaa和bbb都完成之后再发出成功 : function aaa(){ var dfd = \$.Deferred(); dfd.resolve(); return dfd; } function bbb(){ var dfd = \$.Deferred(); dfd.resolve(); //dfd.reject(); return dfd; } \$.when(aaa(), bbb()).done(function(){ //等aaa,bbb都完成之后才会走 alert('成功'); }).fail(function(){ //只要一个失败就走失败 , alert('失败'); });</pre> 	<pre>// Deferred helper when: function(subordinate /* , ..., subordinateN */) { var i = 0, resolveValues = core_slice.call(arguments), length = resolveValues.length, // the count of uncompleted subordinates remaining = length !== 1 (subordinate && jQuery.isFunction(subordinate.promise)) ? length : 0, // the master Deferred. If resolveValues consist of only a single Deferred, just use that. deferred = remaining === 1 ? subordinate : jQuery.Deferred(), // Update function for both resolve and progress values updateFunc = function(i, contexts, values) { return function(value) { contexts[i] = this; values[i] = arguments.length > 1 ? core_slice.call(arguments) : value; if(values === progressValues) { deferred.notifyWith(contexts, values); } else if (!(--remaining)) { deferred.resolveWith(contexts, values); } }; }, progressValues, progressContexts, resolveContexts; // add listeners to Deferred subordinates; treat others as resolved if (length > 1) { progressValues = new Array(length); progressContexts = new Array(length); resolveContexts = new Array(length); for (; i < length; i++) { if (resolveValues[i] && jQuery.isFunction(resolveValues[i].promise)) { resolveValues[i].promise() .done(updateFunc(i, resolveContexts, resolveValues)) .fail(deferred.reject) .progress(updateFunc(i, progressContexts, progressValues)); } else { --remaining; } } } }</pre>



when里面的参数都是延迟对象，如果不是延迟对象，是会被跳过的，不传参数会直接走成功。
写普通对象的作用：传参：
\$.when(123, 123).done(function(){
 alert(arguments[1]);
});

i是参数的总个数，然后将arguments转成数组，然后获得数组长度length，remaining是计数器，然后创建的是整体的deferred对象。

jQuery.isFunction(subordinate.promise)：判断传入的参数是不是延迟对象，

updateFunc：作用就是在done中减计数器，减到0的时候就触发deferred的resolve。

```
        } else {  
            --remaining;  
        }  
    }  
  
    // if we're not waiting on anything, resolve the master  
    if ( !remaining ) {  
        deferred.resolveWith( resolveContexts, resolveValues );  
    }  
  
    return deferred.promise();  
});
```

(3184, 3295) =====

(3184, 3295) support：功能检测 miaov 39~42
support在这个版本中做的处理不多。多的是在1.11等版本中，做IE678的兼容处理。
for(var attr in \$.support){
 \$("body").append('<div>' + attr + ':' + \$.support[attr] + '</div>');
}

然后呢，会得到这么个结果（不同浏览器的truefalse不一样）

CheckOn : true
optSelected : false
reliableMarginRight : true
boxSizingReliable : false
pixelPosition : true
noCloneChecked : true
optDisabled : true
radioValue : false
checkClone : true
focusinBubbles : true
clearCloneStyle : false
cors : true
ajax : true
boxSizing : true

support只是用于检测，hooks来解决兼容问题

(3184) 创建一些元素，通过元素的兼容表现来看是否支持一些功能。	<pre>jQuery.support = (function(support) { var input = document.createElement("input"), fragment = document.createDocumentFragment(), div = document.createElement("div"), select = document.createElement("select"), opt = select.appendChild(document.createElement("option"));</pre>
(3192) 这个没太大必要，最新版本中已经去掉了这个判断，因为input默认会是text，所有浏览器都会走后面。	<pre>// Finish early in limited environments if (!input.type) { return support; }</pre>
(3196) 将input改成了复选框	<pre>input.type = "checkbox";</pre>
(3200) 看复选框的默认value值到底是什么，默认大部分情况下是" on "，但是在老版本的webkit下是空的。那么，在老版本下就要做处理让它默认是on了，这个处理的代码在4292行	<pre>// Support: Safari 5.1, iOS 5.1, Android 4.x, Android 2.3 // Check the default checkbox/radio value (" on" on old WebKit; "on" elsewhere) support.checkOn = input.value !== "";</pre>
(3204) opt.selected是下拉菜单的子项，返回true，说明被选中了，返回false，说明没有被选中。 火狐，chrome下，创建一个下拉菜单，默认情况下，第一个子项是被选中的，但是在IE下是不选中的。	<pre>// Must access the parent to make an option select properly // Support: IE9, IE10 support.optSelected = opt.selected;</pre>
(3207) 定义了初始值 有些是页面加载完就可以判断的，有些需要做dom操作再进行判断。	<pre>// Will be defined later support.reliableMarginRight = true; support.boxSizingReliable = true; support.pixelPosition = false;</pre>
(3213) 让复选框选中，然后克隆一份，看是否被选中，IE9，10都是false，其他是true，让所有浏览器下复制出来的checkbox都能是选中的。	<pre>// Make sure checked status is properly cloned // Support: IE9, IE10 input.checked = true; support.noCloneChecked = input.cloneNode(true).checked;</pre>
(3218) 现在好像都是true，只有在老版本的webkit下才会有影响。	<pre>// Make sure that the options inside disabled selects aren't marked as disabled // (WebKit marks them as disabled) select.disabled = true; support.optDisabled = !opt.disabled;</pre>
(3223) 重新创建了一个input，这个一定要先设置value，再设置type。 IE9，10，11都是false。其他会返回on。	<pre>// Check if an input maintains its value after becoming a radio // Support: IE9, IE10 input = document.createElement("input"); input.value = "t"; input.type = "radio"; support.radioValue = input.value === "t";</pre>
(3229) 判断clone出来的节点是否有属性 IE 火狐 Chrome都是true，Safari 是false，因为Safari的webkit版本低	<pre>// #11217 - WebKit loses check when the name is after the checked attribute input.setAttribute("checked", "t");</pre>

	<pre>input.setAttribute("name", "t"); fragment.appendChild(input); // Support: Safari 5.1, Android 4.x, Android 2.3 // old WebKit doesn't clone checked state correctly in fragments support.checkClone = fragment.cloneNode(true).cloneNode(true).lastChild.checked; (3240) onfocusin在IE下支持，只有FF.Chrome和Safari不支持，这个事件能冒泡，onfocus不能冒泡 // Support: Firefox, Chrome, Safari // Beware of CSP restrictions (https://developer.mozilla.org/en/Security/CSP) support.focusinBubbles = "onfocusin" in window; (3242) 克隆出来的div的修改不应该影响到原div，但是如果影响到了，这里就会返回false。在IE下都返回false，其他返回true。 div.style.backgroundClip = "content-box"; div.cloneNode(true).style.backgroundClip = ""; support.clearCloneStyle = div.style.backgroundClip === "content-box"; (3247) 创建一些DOM节点，然后再进行判断，box-sizing 设置标准模式或者是怪异模式。 divReset 设置标准样式。 找到body标签，判断其是否存在，如果不存在，就直接return了， // Run tests that need a body at doc ready jQuery(function() { var container, marginDiv, // Support: Firefox, Android 2.3 (Prefixed box-sizing versions). divReset = "padding:0;margin:0;border:0;display:block;-webkit-box-sizing:content-box;-moz-box-sizing:content-box;box-sizing:content-box", body = document.getElementsByTagName("body")[0]; if (!body) { // Return for frameset docs that don't have a body return; } container = document.createElement("div"); container.style.cssText = "border:0;width:0;height:0;position:absolute;top:0;left:-9999px;margin-top:1px"; // Check box-sizing and margin behavior. body.appendChild(container).appendChild(div); div.innerHTML = ""; // Support: Firefox, Android 2.3 (Prefixed box-sizing versions). div.style.cssText = "-webkit-box-sizing:border-box;-moz-box-sizing:border-box;box-sizing:border-box;padding:1px;border:1px;display:block;width:4px;margin-top:1%;position:absolute;top:1%"; // Workaround failing boxSizing test due to offsetWidth returning wrong value // with some non-1 values of body zoom, ticket #13543 jQuery.swap(body, body.style.zoom != null ? { zoom: 1 } : {}, function() { support.boxSizing = div.offsetWidth === 4; }); (3274) 在node.js下是没有这个属性的，就不会走if。 ❓ 为什么要在nodejs下进行判断呢？ 里面是像素的position的判断，首先获取到top值，FFChrome IE是true，Safari是false 那就是说，如果你在一个样式的值上设置了百分比，那除了Safari以外，其他浏览器都会把这个百分比自动的转换为像素。 boxSizingReliable：IE下怪异模式，并还有padding时，将得到width = 2px。 创建一个div，判断marginright reliableMarginRight 都是true。 然后把创建的元素删除掉。 --- cors ajax</pre>
	<pre>// Use window.getComputedStyle because jsdom on node.js will break without it. if (window.getComputedStyle) { support.pixelPosition = (window.getComputedStyle(div, null) {}).top !== "1%"; support.boxSizingReliable = (window.getComputedStyle(div, null) { width: "4px" }).width === "4px"; // Support: Android 2.3 // Check if div with explicit width and no margin-right incorrectly // gets computed margin-right based on width of container. (#3333) // WebKit Bug 13343 - getComputedStyle returns wrong value for margin-right marginDiv = div.appendChild(document.createElement("div")); marginDiv.style.cssText = div.style.cssText = divReset; marginDiv.style.marginRight = marginDiv.style.width = "0"; div.style.width = "1px"; support.reliableMarginRight = !parseFloat((window.getComputedStyle(marginDiv, null) {}).marginRight); body.removeChild(container); }; return support; })({});</pre>

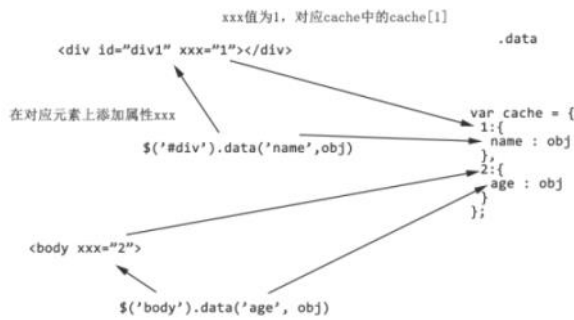
(3308, 3652) =====

(3308, 3652) data()：数据缓存，避免大数据添加到元素身上 造成内存泄露，miaov43~46
首先看看使用情况：
attr(), prop(), data()

```
$('#div1').attr('name', 'hello');
alert( $('#div1').attr('name') );
    document.getElementById('div1').setAttribute('name', 'hello');
    document.getElementById('div1').getAttribute('name');
---
$('#div1').prop('name', 'hello');
alert( $('#div1').prop('name') );
    document.getElementById('div1')['name'] = 'hello';
    document.getElementById('div1')['name'];
---
$('#div1').data('name', 'hello');
alert( $('#div1').data('name') );
以上三个执行结果都是一样的。
attr prop 设置元素本身的属性较适合。
data可以加载大量数据。
```

内存泄露：DOM元素与对象之间互相引用，大部分浏览器就会出现内存泄露
var oDiv = document.getElementById('div1');
var obj = {};
oDiv.name = obj;
obj.age = oDiv;

简单原理：



简化版本:

```
jQuery.extend(  
  acceptData  
  hasData  
  data  
  removeData  
  _data  
  _removeData  
);  
jQuery.fn.extend(  
  data  
  removeData  
);
```

简单用法：

```
$('#div').data('name', 'hello');  
// $('#div').removeData('name');  
alert( $('#div').data('name') )  
---  
$.data(document.body, 'age', 30);  
$.hasData(document.body, 'age');  
$.removeData(document.body, 'age');  
alert( $.data(document.body, 'age') );
```

Data对象下的方法：

```
Data.prototype = {  
  key  
  set  
  get  
  access  
  remove  
  hasData  
  discard  
}
```

	<pre>/* Implementation Summary 1. Enforce API surface and semantic compatibility with 1.9.x branch 2. Improve the module's maintainability by reducing the storage paths to a single mechanism. 3. Use the same single mechanism to support "private" and "user" data. 4. _Never_ expose "private" data to user code (TODO: Drop _data, _removeData) 5. Avoid exposing implementation details on user objects (eg. expando properties) 6. Provide a clear path for implementation upgrade to WeakMap in 2014 */ var data_user, data_priv, rbrace = /(?:\{[\s\S]*\} \[[\s\S]*\])\$/, rmultiDash = /[A-Z]/g;</pre>
<p>(3308) data_user是用来存数据的，获取设置方法：\$.data(el,key,value) data_priv是用来存事件的，click事件那种，\$.data(el, key, value) data_user和data_priv, 就如其名, 一个是用户用的, 一个是jQuery私有的, 他们都是一个叫Data的实例对象</p>	<pre>function Data() { // Support: Android < 4, // Old WebKit does not have Object.preventExtensions/freeze method, // return new empty object instead with no [[set]] accessor Object.defineProperty(this.cache = {}, 0, { get: function() { return {}; } }); this.expando = jQuery.expando + Math.random(); }</pre>
<p>(3312) Data() freeze 冻结，防止修改对象。 defineProperty 是ECMA5的一个方法，第一个参数是一个对象，第二个是一个属性值(0 : {}), 第三个只写了get, 说明只能获取，不能设置: Object.defineProperty(obj, 0 { get: function(){ return {}; } }); alert(obj[0]); obj[0] = 123; //没有修改成功 alert(obj[0]); 0是一个公用的，凡是不是满足条件的元素（ 3334 ），都会分配0, 1、 2、 3是私有的，是指定某一个元素才有的。 this.expando唯一标识，就是元素上的那个属性xxx, 上图有标明的。</p>	<pre>Data.uid = 1; Data.accepts = function(owner) { // Accepts only: // - Node // - Node.ELEMENT_NODE // - Node.DOCUMENT_NODE // - Object // - Any return owner.nodeType ? owner.nodeType === 1 owner.nodeType === 9 : true; };</pre>
<p>(3325) uid就是可以累加的一个属性值，取值为123456，就是cache中的属性值。 Data.accepts 判断节点类型。 1为元素element，9为文档document document.nodeType = 9 其他的标签节点为1 document.body.nodeType = 1</p>	

<p>(3339) key : 获取到的就是cache中的属性值 (123456那种) 。</p> <p>首先判断节点是否符合条件，不符合的直接跳过。</p> <p>然后找一下元素节点身上是不是有 xxx那个属性值 (this.expando) 。</p> <p>第一次是找不到的，unlock=undefined，所以要分配一个；</p> <p>使用defineProperties，给元素身上添加值，让它只能获取不能设置， ? 这里没有对值的获取设置做限制呢？ 但这种方式安卓<4不支持，所以还是要用传统的方法添加。</p> <p>设置一个cache，json，unlock就是this.cache中的属性 (123456那种)，即是key值。</p> <p>可以这么理解，如果要找的owner在this.cache中已经有坑了，将返回其站的坑的key值，否则将创建一个空的json对象，然后返回其占的坑的key值。</p>	<pre>Data.prototype = { key: function(owner) { // We can accept data for non-element nodes in modern browsers, // but we should not, see #8335. // Always return the key for a frozen object. if (!Data.accepts(owner)) { return 0; } var descriptor = {}, // Check if the owner object already has a cache key unlock = owner[this.expando]; // If not, create one if (!unlock) { unlock = Data.uid++; // Secure it in a non-enumerable, non-writable property try { descriptor[this.expando] = { value: unlock }; Object.defineProperties(owner, descriptor); // Support: Android < 4 // Fallback to a less secure definition } catch (e) { descriptor[this.expando] = unlock; jQuery.extend(owner, descriptor); } } // Ensure the cache object if (!this.cache[unlock]) { this.cache[unlock] = {}; } return unlock; }, </pre>
<p>(3375) set</p> <p>通过key方法，找到对象owner占坑的key值 (123456那种)，然后再通过缓存找到对应的json。</p> <p>然后判断添加的属性是不是字符串，不是字符串走else else中处理： \$.data(document.body, {age: 30, 'job': 'it'}) 这种</p> <p>这个其实判断是不是空没有必要，因为最后两个都是使用的是forin</p>	<pre>set: function(owner, data, value) { var prop, // There may be an unlock assigned to this node, // if there is no entry for this "owner", create one inline // and set the unlock as though an owner entry had always existed unlock = this.key(owner), cache = this.cache[unlock]; // Handle: [owner, key, value] args if (typeof data === "string") { cache[data] = value; // Handle: [owner, { properties }] args } else { // Fresh assignments by object are shallow copied if (jQuery.isEmptyObject(cache)) { jQuery.extend(this.cache[unlock], data); } // Otherwise, copy the properties one-by-one to the cache object } else { for (prop in data) { cache[prop] = data[prop]; } } return cache; }, </pre>
<p>(3401) get 获取，</p> <p>先找到key，再找到cache，再获得cache下面的所有值</p> <p>如果只传了一个参数，将得到所有的cache值，如果传了2个参数将只得指定的cache值</p>	<pre>get: function(owner, key) { // Either a valid cache is found, or will be created. // New caches will be created and the unlock returned, // allowing direct access to the newly created // empty data object. A valid owner object must be provided. var cache = this.cache[this.key(owner)]; return key === undefined ? cache : cache[key]; }, </pre>
<p>(3411) access : 对get和set进行整合</p>	<pre>access: function(owner, key, value) { var stored; // In cases where either: // 1. No key was specified // 2. A string key was specified, but no value provided // Take the "read" path and allow the get method to determine // which value to return, respectively either: // 1. The entire cache object // 2. The data stored at the key if (key === undefined ((key && typeof key === "string") && value === undefined)) { stored = this.get(owner, key); return stored !== undefined ? stored : this.get(owner, jQuery.camelCase(key)); } // [*]When the key is not a string, or both a key and value // are specified, set or extend (existing objects) with either: // 1. An object of properties // 2. A key and value this.set(owner, key, value); // Since the "set" path can have two possible entry points // return the expected data based on which path was taken[*] return value !== undefined ? value : key; }, </pre>
<p>(3445) remove</p> <p>如果不指定key值，那么会把元素下面所有的属性值都清空</p> <p>如果传递了值，然后判断参数是不是数组：</p>	<pre>remove: function(owner, key) { var i, name, camel, unlock = this.key(owner), cache = this.cache[unlock]; if (key === undefined) { this.cache[unlock] = {}; } else { // If key is a string, push it onto the list of keys // and delete the key from the cache data. // If key is an array, delete the entire cache if (typeof key === "string") { cache[key] = undefined; } else { delete cache; } } } </pre>

<pre>\$.removeData(document.body, ['age', 'job']);</pre> <p>这里做了处理要返回驼峰的写法，就是存储的值中基本上是以驼峰命名的，除非出现了冲突</p>	<pre>// Support array or space separated string of keys if (jQuery.isArray(key)) { // If "name" is an array of keys... // When data is initially created, via ("key", "val") signature, // keys will be converted to camelCase. // Since there is no way to tell how a key was added, remove // both plain key and camelCase key. #12786 // This will only penalize the array argument path. name = key.concat(key.map(jQuery.camelCase)); } else { camel = jQuery.camelCase(key); // Try the string as a key before any manipulation if (key in cache) { name = [key, camel]; } else { // If a key with the spaces exists, use it. // Otherwise, create an array by matching non-whitespace name = camel; name = name in cache ? [name] : (name.match(core_rnotwhite) []); } } i = name.length; while (i--) { delete cache[name[i]]; } },</pre>
<p>(3483) has 做判断</p>	<pre>hasData: function(owner) { return !jQuery.isEmptyObject(this.cache[owner[this.expando]] {}); },</pre>
<p>(3488) discard，删除一个整体对象，不是个别的属性</p>	<pre>discard: function(owner) { if (owner[this.expando]) { delete this.cache[owner[this.expando]]; } } };</pre>
<p>(3496) 看到前面的3308行</p>	<pre>// These may be used throughout the jQuery core codebase data_user = new Data(); data_priv = new Data();</pre>
<p>(3500) 此处就是提供给外部的接口。\$.data 这种用法，可以理解是工具方法</p> <p>可以看到，只要是对象的数据的，都使用的是data_user进行的处理</p> <p>事件的是内部使用的，都是data_priv进行的处理</p>	<pre>jQuery.extend({ acceptData: Data.accepts, hasData: function(elem) { return data_user.hasData(elem) data_priv.hasData(elem); }, data: function(elem, name, data) { return data_user.access(elem, name, data); }, removeData: function(elem, name) { data_user.remove(elem, name); }, // TODO: Now that all calls to _data and _removeData have been replaced // with direct calls to data_priv methods, these can be deprecated. _data: function(elem, name, data) { return data_priv.access(elem, name, data); }, _removeData: function(elem, name) { data_priv.remove(elem, name); } });</pre>
<p>(3526) \$.data 这种用法，可以理解是实例的方法</p> <p>jq的设计小思想：</p> <p>如果是设置一组元素的值，那么会设置这一组元素的每一个值。</p> <p>如果是获取一组元素的值，那么只会获取到第一个元素的值。</p> <p>这个data函数有两个参数，当写一个key值的时候就是获取，写两个参数的时候就是设置值。</p> <p>elem存的就是获取的时候，第一个元素。</p> <p>当key值为空的时候，会返回所有的值。</p> <p>this.length 判断有没有这个元素</p> <p>然后通过get方法来获取到数据。</p> <p>if判断针对于：</p> <pre><div id = "div1" data-miaov = "miaov">aaa</div></pre> <p>返回数据中会包含html5设置的数据数据值 miaov = miaov</p> <p>attrs = elem.attributes; 获取元素所有属性的一个集合，然后判断获得的这些属性中是否包含前缀有data-的，</p>	<pre>jQuery.fn.extend({ data: function(key, value) { var attrs, name, elem = this[0], i = 0, data = null; // Gets all values if (key === undefined) { if (this.length) { data = data_user.get(elem); if (elem.nodeType === 1 && !data_priv.get(elem, "hasDataAttrs")) { attrs = elem.attributes; for (; i < attrs.length; i++) { name = attrs[i].name; if (name.indexOf("data-") === 0) { name = jQuery.camelCase(name.slice(5)); dataAttr(elem, name, data[name]); } } data_priv.set(elem, "hasDataAttrs", true); } } return data; } } });</pre>
<p>(3556) 判断是不是设置多个值的情况。比如下面这种：</p> <pre>\$('#div').data({ name: 'hello', age: 30 })</pre> <p>对每个元素进行遍历，然后通过set去设置值。</p> <p>这个返回的时候，access是对多功能值的设置，获取，以及回调的操作。</p> <p>先判断这个value是否为空，如果是，走if，这个if里面的操作都是获取。</p>	<pre>// Sets multiple values if (typeof key === "object") { return this.each(function() { data_user.set(this, key); }); } return jQuery.access(this, function(value) { var data, camelKey = jQuery.camelCase(key); // The calling jQuery object (element matches) is not empty // (and therefore has an element appears at this[0]) and the // `value` parameter was not undefined. An empty jQuery object // will result in `undefined` for elem = this[0] which will // throw an exception if an attempt to read a data cache is made. if (elem && value === undefined) { // Attempt to get data from the cache // with the key as-is data = data_user.get(elem, key); }</pre>

<p>这种是获取最普遍的情况，即元素组合中存在这个key值，直接获取返回。</p> <p>接下来呢，对这个key值进行转驼峰，转完驼峰再去找，有就返回。</p> <p>如果还没有找到呢，那就去找html5中是否定义了这个属性。找到就返回。</p> <p>设置值，就是每一个值进行遍历。 \$('#div').data('nameAge', 'hi'); \$('#div').data('name-age', 'hello'); this.cache = { 1 : { 'nameAge': 'hello', 'name-age': 'hello' } } 当arguments.length = 1时，是false，进行获取操作，否则就是设置操作。</p>	<pre>if (data !== undefined) { return data; } // Attempt to get data from the cache // with the key camelized data = data_user.get(elem, camelKey); if (data !== undefined) { return data; } // Attempt to "discover" the data in // HTML5 custom data-* attrs data = dataAttr(elem, camelKey, undefined); if (data !== undefined) { return data; } // We tried really hard, but the data doesn't exist. return; } // Set the data... this.each(function() { // First, attempt to store a copy or reference of any // data that might've been store with a camelCased key. var data = data_user.get(this, camelKey); // For HTML5 data-* attribute interop, we have to // store property names with dashes in a camelCase form. // This might not apply to all properties... data_user.set(this, camelKey, value); // *... In the case of properties that might _actually_ // have dashes, we need to also store a copy of that // unchanged property. if (key.indexOf("-") !== -1 && data !== undefined) { data_user.set(this, key, value); } }); }, null, value, arguments.length > 1, null, true); },</pre>
<p>(3618) removeData 遍历每一个，调用remove的方法</p>	<pre>removeData: function(key) { return this.each(function() { data_user.remove(this, key); }); } });</pre>
<p>(3625) dataAttr，获取到html5中通过data-定义的数据的值</p> <p>要判断在data中是否已经存在需要查找的元素： <div id = "div1" data-miaov = "miaov">aaa</div> \$('#div1').data("miaov", "aaa"); 像这种，html5的data-和data()都进行了设置，那么miaov的值会是aaa rmultiDash这个正则就是把字符串中大写字母转成小写，比如miaovAll -> data-miaov-all。 然后判断这个字符串的类型，就是说如果你在html中写的true，false，null，那么这里就是对应的类型值，不会给你"false""true""null"。 +data 转成数字在进行比较，即把字符串转成数字存储。 rbrace.test(data) 判断是不是json格式，然后转成json 如果都不是呢，就直接存的是data</p>	<pre>function dataAttr(elem, key, data) { var name; // If nothing was found internally, try to fetch any // data from the HTML5 data-* attribute if (data === undefined && elem.nodeType === 1) { 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 : // Only convert to a number if it doesn't change the string +data + "" === data ? +data : rbrace.test(data) ? JSON.parse(data) : data; } catch(e) {} // Make sure we set the data so it isn't changed later data_user.set(elem, key, data); } else { data = undefined; } } return data; }</pre>

(3653, 3797) =====

```
( 3653, 3797 ) queue() : 队列管理，执行顺序管理 miaov47~50  
精简的结构  
jQuery.extend(  
    queue        push()  
    dequeue      shift()  
    _queueHooks  
)  
jQuery.fn.extend(  
    queue  
    dequeue  
    delay  
    clearQueue  
    promise  
)  
用法：  
function aaa(){  
    alert(1);  
}  
function bbb(){  
    alert(2);  
}  
$.queue(document, 'q1', aaa); // 在document下建立了一个q1的队列，里面添加了一个aaa的函数。  
$.queue(document, 'q1', bbb);  
// $.queue(document, 'q1', [aaa, bbb]);  
传的必须是函数。队列中存储的都是函数。  
$.queue(document, 'q1'); //aaa() 取出第一个函数，并执行  
$.queue(document, 'q1'); //bbb()
```



```

-----
$(document).queue('q1', aaa);
$(document).queue('q1', bbb);
$(document).dequeue('q1'); //aaa
$(document).dequeue('q1'); //bbb
---
主要还是针对内部的运动使用的
$('#div').click(function(){
    $(this).animate({ width : 300, 2000);
    $(this).animate({ height:300, 2000);
    $(this).animate({ left:300, 2000);
});
会一步一步的动，执行时长是6s。
---
动画使用的队列的名字是 'fx' :
$('#div').click(function(){
    $(this).animate({ width : 300, 2000).queue('fx', function() { //这里fx不写也可以。
        $(this).dequeue(); //队列中要有出队操作。
    }).animate({ left:300, 2000);
});

```

<p>(3654) queue</p> <p>接收单个参数，对象，队列名字，要往队列中添加的数据。</p> <p>判断对象是否存在</p> <p>队列中默认的就是运动的队列的名字，</p> <p>首先取一下queue，</p> <p>如果queue不存在，则创建一个数据缓存，</p> <p>还判断了data是不是数组，如果是数组，那会把之前的去掉，只添加当前数组：</p> <pre> \$.queue(document, 'q1', aaa); \$.queue(document, 'q1', [bbb]); //现在只有bbb 有了之后就push </pre>	<pre> jQuery.extend({ queue: function(elem, type, data) { var queue; if (elem) { type = (type "fx") + "queue"; queue = data_priv.get(elem, type); // Speed up dequeue by getting out quickly if this is just a lookup if (data) { if (!queue jQuery.isArray(data)) { queue = data_priv.access(elem, type, jQuery.makeArray(data)); } else { queue.push(data); } } return queue []; } }, </pre>
<p>(3673) dequeue</p> <p>首先判断下type类型，默认的就是fx，</p> <p>首先得到queue这个队列，然后得到队列的一个长度，</p> <p>然后得到这个先添加进去的函数，fn，从数组头部取出，</p> <p>next中做一个出队操作，</p> <p>inprogress 是专门针对fx队列的，</p> <p>//这里就相当于找到aaa这个函数，然后让他执行next传参</p>	<pre> dequeue: function(elem, type) { type = type "fx"; var queue = jQuery.queue(elem, type), startLength = queue.length, fn = queue.shift(), hooks = jQuery._queueHooks(elem, type), next = function() { jQuery.dequeue(elem, type); }; // If the fx queue is dequeued, always remove the progress sentinel if (fn === "inprogress") { fn = queue.shift(); startLength--; } if (fn) { // Add a progress sentinel to prevent the fx queue from being // automatically dequeued if (type === "fx") { queue.unshift("inprogress"); } // clear up the last queue stop function delete hooks.stop; fn.call(elem, next, hooks); } if (!startLength && hooks) { hooks.empty.fire(); } }, </pre>
<p>(3709) 不管进来的是什么，都能进行操作</p> <p>出队结束后，主动触发remove操作，清理一下缓存操作。</p>	<pre> // not intended for public consumption - generates a queueHooks object, or returns the current one _queueHooks: function(elem, type) { var key = type + "queueHooks"; return data_priv.get(elem, key) data_priv.access(elem, key, { empty: jQuery.Callbacks("once memory").add(function() { data_priv.remove(elem, [type + "queue", key]); }) }); } </pre>
<p>(3720) 实例方法queue</p> <p>判断type类型，看是否是字符串，如果不是，默认为fx。</p> <p>根据arguments的长度判断当前应该是设置还是获取，</p> <pre> \$(document).queue('q1'); </pre> <p>值返回一组当中的第0个。</p> <p>设置的时候是对每一项都进行设置。</p> <p>(3742) 这里是入队方法，却调用了出队的方法，针对的就是运动。</p> <p>直接出队不行么？只针对第一次的运动进行出队操作。</p>	<pre> jQuery.fn.extend({ queue: function(type, data) { var setter = 2; if (typeof type !== "string") { data = type; type = "fx"; setter--; } if (arguments.length < setter) { return jQuery.queue(this[0], type); } return data === undefined ? this : this.each(function() { var queue = jQuery.queue(this, type, data); // ensure a hooks for this queue jQuery._queueHooks(this, type); if (type === "fx" && queue[0] !== "inprogress") { jQuery.dequeue(this, type); } }); }, </pre>
<p>(3746) 对每一个进行遍历，然后取调用工具方法的dequeue</p>	<pre> dequeue: function(type) { return this.each(function() { jQuery.dequeue(this, type); }); } </pre>

	<pre> }, </pre>
<p>(3753) delay , 让队列进行暂停 , 即延迟队列执行</p> <p><code>\$('#div1').animate({width:300}, 2000).delay(2000).animate({left:300}, 2000);</code></p> <p>jQuery.fx.speeds 是可以传参传文字的 , delay('slow') //fast</p> <p>stop是清定时器 , 一般用不上 ,</p>	<pre> // Based off of the plugin by Clint Helfers, with permission. // http://blindsignals.com/index.php/2009/07/jquery-delay/ delay: function(time, type) { time = jQuery.fx ? jQuery.fx.speeds[time] time : time; type = type "fx"; return this.queue(type, function(next, hooks) { var timeout = setTimeout(next, time); hooks.stop = function() { clearTimeout(timeout); }; }); }, </pre>
<p>(3764) 清除队列 , 让队列设置成空数组</p>	<pre> clearQueue: function(type) { return this.queue(type "fx", []); }, </pre>
<p>(3769) promise , 等整个队列结束之后就触发</p> <p><code>\$('#div1').click(function(){</code> <code>\$(this).animate({width:300}, 2000).animate({left:300}, 2000);</code> <code>\$(this).promise().done(function(){ //在队列全部执行之后再去调用</code> <code> alert(123);</code> <code>});</code> <code>});</code></p> <p>count计数有多少个需要执行的队列。</p> <p>当执行完 , 出队操作执行到0时就完成了 ,</p> <p>当队列存在的时候就累加。</p>	<pre> // Get a promise resolved when queues of a certain type // are emptied (fx is the type by default) promise: function(type, obj) { var tmp, count = 1, defer = jQuery.Deferred(), elements = this, i = this.length, resolve = function() { if (!(--count)) { defer.resolveWith(elements, [elements]); } }; if (typeof type !== "string") { obj = type; type = undefined; } type = type "fx"; while(i--) { tmp = data_priv.get(elements[i], type + "queueHooks"); if (tmp && tmp.empty) { count++; tmp.empty.add(resolve); } } resolve(); return defer.promise(obj); } }); </pre>
<p>(3803, 4299) =====</p> <p>(4300, 5128) =====</p> <p>=====</p>	