```
/*
 * 说明:
 * 本源代码的中文注释乃Auscar
lin呕心沥血所作.旨在促进jQuery的传播以及向广大jQuery爱好者提供一个进阶
 *
的途径,以让各位更加深入地了解jQuery,学习其中有用的技术,从而为振兴中华J
S出一份绵薄之力...(说大了...)
 *
 *
本文件保留了jQuery代码原来的英文注释,个别语句我在其旁列出了尝试性的翻
译(并标明这是翻译).水平有限希望读者能斟酌.
 *
另外,本中文注释不是简单将原文翻译(jQuery作者那少得可怜的注释根本不足以
让我们读通jQuery的源代码).
 *
而是尽本人最大的努力将程序的意图以及所涉及的中高级的JavaScript程序设计
技术展现给各位读者,故文件注释较为详尽.
 *
 * 在注释的书写风格方面,采取了比较随意的方式和语气,
目的在于避免晦涩的说教以及拉近读者与代码之间的距离,同时也是为了
 * 增强大家在阅读代码的趣味性.另外,本人并不提倡使用中文进行注释,
但是为了扩大读者群体, 不得已为之...
 *
 * 见识肤浅,不足之处希望指出.
我也希望有人能理解与支持我的工作.如果你觉得我的注释对你有帮助,
请不要吝惜你的感谢~
 * 批评与鼓励还有建议都可以通过以下这个电子邮件地址发送给我:
 * auscar@126.com
 *
 * 或者登录我的个人网站给我留言:
 * http://www.linhuihua.com   (注:linhuihua是我的中文名拼音)
 *
 * 又另外,本人写的仿jQuery js 框架miniQ即将要跟大家见面.
 * 这个框架比jQuery短小,功能也没有这么强大.
但它的架构完全仿照jQuery.可以说它是jQuery架构的一个DEMO.
 *
透过它,你可以从整体上把握jQuery的框架以及所用到的程序技巧和设计模式.可
以说它是一个教学用的小框架.
 *
 * 希望能有越来越多的人喜欢上jQuery,享受jQuery!
 *
 */
/*
 * 版权声明:
 * (1) 本文件中的JavaScript代码与英文注释版权归原作者所有
 * (2) 本文件中的中文注释版权归本人所有. 请自由下载与传播本文件,
但请勿用于商业用途.
 */

/*
 * jQuery 1.2.6 - New Wave Javascript
 *
 * Copyright (c) 2008 John Resig (jquery.com)
 * Dual licensed under the MIT (MIT-LICENSE.txt)
 * and GPL (GPL-LICENSE.txt) licenses.
 *
 * $Date: 2008-05-24 14:22:17 -0400 (Sat, 24 May 2008) $
 * $Rev: 5685 $
```

```
 */


/*
 *
整个jQuery代码都定义在一个自运行(定义完成之后马上运行)的匿名函数的内部:
 * (function(){
 *
 *      //jQuery code runs here
 *
 * })();
 *
 * 这样, 这个匿名函数所形成的闭包会保护jQuery的代码,
避免了匿名函数内部的jQuery代码与外部之间发生冲突(如使用了相同的变量名).
 * 另外, 函数自运行也保证了jQuery在能在第一时间得到初始化.
 */
(function(){
/*
 * 写下面两行代码, 是出于这样的考虑:
 *
在多库共存的环境中, 很可能会与别人的库使用相同的关键字, 那就先把人家的jQuery、$(如果真的有人用的话)保存下来,
 *
然后再换上自己的。需要的时候再把自己的jQuery,$关键字卸掉, 恢复人家的代码对这个两个关键字的使用权.调用jQuery.noConflict便可恢复.
 * 恢复的详细内容, 请参考jQuery.noConflict的中文注释.
 */
// Map over jQuery in case of overwrite
// 保存jQuery的关键字, 免得被重写后没法恢复.
var _jQuery = window.jQuery,
// Map over the $ in case of overwrite
// 保存$的关键字, 免得被重写后没法恢复.
    _$ = window.$;


/* 当前作用域内(也就是这个自运行的匿名函数所形成的闭包内)、
全局作用域内中的jQuery和$都是注释下面的这个函数的引用.
 * 而这个函数实际上是调用jQuery.fn.init来构造一个jQuery对象.
 * 使用jQuery的人可能会奇怪:
为什么我直接使用$('#someId')就能选择到我要的对象? 怎么$就可以用啦?
$到底是什么意思? 这里的代码就能给出一点答案:
 * 原来$不过是一个jQuery构造函数的引用. 使用$()就是调用了下面这个函数.
 */
/**
 * jQuery的构造函数.通过selector选择器构造并返回一个jQuery对象.
 * @param {string} selector
 * @param {Object} context
 */
var jQuery = window.jQuery = window.$ = function( selector, context ) {
    // The jQuery object is actually just the init constructor 'enhanced'
    /*
     * 实际上,jQuery.fn.init才是jQuery真正的构造函数.
这里就是jQuery美妙世界的入口.
     */
```

为什么会出现重写的情况（why was it rewritten?）

```
86      return new jQuery.fn.init( selector, context );
87  };
88
89  // A simple way to check for HTML strings or ID strings
90  // (both of which we optimize for)
91  /* 翻译:一个检测HTML字符串和ID字符串的简单方法.
92   * 说明:
93   * 以下是一个正则表达式,意在快速地检测字符串是不是HTML string or ID
    string
94   */
95  var quickExpr = /^[^<]*(<(.|\s)+>)[^>]*$|^#(\w+)$/,
96
97  // Is it a simple selector
98  //
    isSimple是一个简单选择器的正则表达式.简单是说选择器字符串里面只有一个
    选择器,如:
99  // '#eleID'就是一个简单的选择器,而:
100 // '#eleID .address'则是一个复杂(complex)选择器
101     isSimple = /^.[^:#\[\.]*$/,
102
103 // Will speed up references to undefined, and allows munging its name.
104     undefined;
105 /*
106  * 我要对以上这个undefined进行解释:
107  * ECMAScript
    v3的规范里面规定了名为undefined的全局变量,它的初始值就是undefined.
108  *
    但是我们并不确定世界上的任何一个JavaScript实现都有全局变量undefined,那
    么这时我们
109  * 只需要自己声明一个但不给它赋值即可,它的值就是undefined.
110  *
111  * 上面那个undefined就是声明了这样的一个变量.
112  *
113  */
114
115 /*
116  * 下面的对通过jQuery.prototype的定义来规划jQuery对象的主要行为.
    所有通过构造函数(如jQuery,$)new出来的jQuery对象都继承来自jQuery.protot
    ype
117  * 的属性和方法. 注意jQuery.prototype使用了另外一个别名:jQuery.fn.
    在jQuery的代码当中, 使用jQuery.fn来代替jQuery.prototype, 其实是一样的.
118  */
119 jQuery.fn = jQuery.prototype = {
120     /**
121      * jQuery的初始化函数. 每次new 一个jQuery对象的时候,
    总是首先由这个函数实现初始化.
122      *
    初始化的主要工作是根据选择器selector选择到匹配的元素,并将这些元素放入
    一个jQuery称之为matched element set的集合当中, 最后返回这个jQuery对象.
123      * 注意, jQuery对象并没有实实在在的一个"matched element
    set"属性. 假设我们的新定义了一个jQuery对象:
124      * var jq = new jQuery('a');
125      *
    那么页面上的所有a标签元素就会以jq[0],jq[1],jq[2]...,jq[n]的形式存储到j
    Query对象(即jq)当中. jQuery把这些匹配到的元素在逻辑上看作是一个
126      * 集合, 并称之为"matched element set".
127      *
128      * @param {string} selector - 选择器.
```

So all the methods hang under the prototype.(所以所有的方法都挂在prototype下)

```
    以这个字符串指定需要选择的元素.
129      * @param {Object} context - 选择器的上下文.
    即指明要在一个什么范围之内选择seletor所指定的元素.
130      */
131     init: function( selector, context ) {
132         // Make sure that a selection was provided
133         // 如果没有传入selector,那么docuemnt就会成为默认的selector.
    没有selector就用document来"凑数".让"matched element set"里面至少要
134         // 有一个元素.
135         selector = selector || document;
136
137         /*
138          * 下面要对selecotr对象进行进行分类的检查,不同类型,不同的处理.
139          * selector可能的类型如下:
140          * (1) 直接的一个Dom元素类型
141          * (2) 数组类型
142          * (3) 函数(function)类型
143          * (4) jQuery或者其他的类数组对象类型
144          * (5) string类型
145          *         a) 没有context的情况
146          *         b) 有context的情况
147          */
148
149         /* 好了, 现在就分情况进行处理 */
150
151         // Handle $(DOMElement)
152         // 是不是(1) "直接的一个Dom元素类型" 啊?如果是,
    就将这个Dom元素直接放入jQuery对象的[0]属性中,
    设置匹配元素集合的大小为1, 返回
153         if ( selector.nodeType ) {//是Dom元素就应该有一个nodeType
154             this[0] = selector;
155             this.length = 1;
156             return this;
157         }
158
159         // Handle HTML strings
160         // 是不是类型(5) - string类型?
161         if ( typeof selector == "string" ) {
162             // Are we dealing with HTML string or an ID?
163             // 翻译:我们是否正在处理HTML或者ID字符串?
164             var match = quickExpr.exec( selector );
165
166             /*
167              * 通过match变量来将string类型的情况再区分成两类:
168              * (1) 是HTML字符串或者ID字符串的情况
169              * (2) 其他,如'.className', 'tagName'之类.
170              */
171
172             // Verify a match, and that no context was specified for
    #id
173             // 核对这个匹配, 还有那些没有为#id提供context的情况.
174             if ( match && (match[1] || !context) ) {
175
176                 // HANDLE: $(html) -> $(array)
177                 // 如果传入的是HTML:
178                 //
    那么调用jQuery.clean将字符串转化成真正的DOM元素然后装在一个数组里面,
    最后返回给selector
```

there are two kinds of logical judgment (这里要有两种逻辑判断)

```
                // 这样selector最后将变成了(2)类型.
                if ( match[1] )
                    selector = jQuery.clean( [ match[1] ], context );
//clean的作用就是将传入的HTML string转化成为DOM

//元素, 并用一个数组装着,最后返回.

                // HANDLE: $("#id")
                // 如果传入的是#id
                else {
                    /* 如果是#id,
那就调用JavaScript原生的getElementById
                     *
有些jQuery的性能提升方法当中建议尽量使用id选择符, 说这样比较高效.
从这里可以看到, 是有道理的.
                     */
                    var elem = document.getElementById( match[3] );

                    // Make sure an element was located
                    // 翻译: 确保一个元素被定位. 即能够get到一个元素
                    if ( elem ){

                        /*
                         * 原本可以直接返回结果了,
但是由于IE和Opera有一个小小的Bug, 因此要处理一下再返回.
                         */

                        // Handle the case where IE and Opera return
items
                        // by name instead of ID
                        // COMP: 翻译:
处理IE和Opera会用name代替ID返回元素的问题
                        if ( elem.id != match[3] )

//jQuery()将会返回一个用document生成的jQuery对象
                            return jQuery().find( selector );

                        // Otherwise, we inject the element directly
into the jQuery object
                        // 好了,
我们将选择到的元素注入到jQuery对象的里, 最后返回.
                        return jQuery( elem );
                    }
                    // 如果代码能运行到这里,
说明match[3]并不能让getElementById选择到任何元素,
把selector设置成[], 让后面的代码收拾手尾
                    selector = [];
                }


            }

            // HANDLE: $(expr, [context])
            // (which is just equivalent to: $(content).find(expr)
            // 翻译: 处理 $(expr,[context])
            // (这跟$(content).find(expr)是一样的)
            /*
如果传入的selector不是HTML字符串或者ID字符串(如'.class','div'之类),那
```

```
就用context新建一个jQuery对象, 然后在这个对象中再查找
223             * selector所指定的元素, 最后返回一个jQuery对象.
更多细节, 可以参考jQuery.fn.find函数的中文注释.
224             */
225            else
226                return jQuery( context ).find( selector );
227        }
228
229
230
231        // HANDLE: $(function)
232        // Shortcut for document ready
233        // 看看是不是类型(3)- 函数(function)类型?
234        // 如果传进来的是一个function,
那么就用docuemnt新建一个jQuery对象, 让后使用jQuery对象的ready函数,
将selector(现在它是一个函数)绑定
235        // 到DOM Ready 事件(不知道什么是DOM Ready事件? Ctrl+F搜索
"read:"!最后一个搜索结果有说明 ).
236        // 要完成上面我所说的功能下面代码中的'load'就显得比较诡异.
在 jQuery 1.3.2里面, 下面的代码已经被改进为:
237        /*
238         * else if ( jQuery.isFunction( selector ) )
239         *     return jQuery( document ).ready( selector );
240         */
241        else if ( jQuery.isFunction( selector ) )
242            return jQuery( document )[ jQuery.fn.ready ? "ready" :
"load" ]( selector );
243
244
245        // 如果是类型(2)和(4)(从代码中可以看出他们是作相同的处理的):
246        //
经过上面的处理之后,程序还能运行到这里(没有return),说明selector是一个类
数组对象(或者就是Array对象). 那我们就不管你是jQuery对象还是Array对
247        // 象,都使用
jQuery.makeArray(selector)来把这个selector对象转换成一个真正的Array.
248        //
最后使用setArray将数组放到自己(也就是this,它是一个jQuery对象)存储匹配
元素的数组(matched element set)里面,然后返回自己这个jQuery对象
249        // 有关细节, 可以参考jQuery.fn.setArray,
jQuery.fn.setArray的中文注释.
250        return this.setArray(jQuery.makeArray(selector));
251    },
252
253
254    // The current version of jQuery being used
255    /*
256     * jQuery当前版本。
257     * 有些代码会检测这个属性来确定对象是不是jQuery对象
258     */
259    jquery: "1.2.6",
260
261    // The number of elements contained in the matched element set
262    /**
263     *
返回匹配元素集合的大小,就是说通过选择器现在到底选择中了多少个元素。这
个数字存在length里面
264     *
这里所说的"匹配元素集合"是指存在每一个jQuery对象中的一个数组。当我们用
```

```
一个选择器创建一个jQuery
265          *
对象时，选择器选中的所有匹配的元素就会存在这个数组里面。如jQuery("div"
), 文档中所有div的引用都
266          * 会被保存在这个数组里面。
267          *
268          * 以下的注释里面都会沿用"匹配元素集合"这个概念。
269          */
270         size: function() {
271             return this.length;
272         },
273
274         // The number of elements contained in the matched element set
275         // 匹配元素集合长度,初始设置为0
276         length: 0,
277
278         // Get the Nth element in the matched element set OR
279         // Get the whole matched element set as a clean array
280         /*
281          * 取得匹配元素集合中的第N个元素
282          */
283         get: function( num ) {
284             return num == undefined ?
285
286                 // Return a 'clean' array
287                 //
如果传进来的num是一个未定义的值，返回一个"干净"的数组.
意思就是返回一个新建的数组副本
288                 jQuery.makeArray( this ) :
289
290                 // Return just the object
291                 // 如果num有值，直接就返回num这个位置上的那个元素
292                 this[ num ];
293         },
294
295         // Take an array of elements and push it onto the stack
296         // (returning the new matched element set)
297         /*
298          *
使用传入的元素生成一个新的jQuery元素,并将这个对象的prevObject设置成当
前这个对象(this).最后将这个新生成的jQuery对象返回.
299          * 似乎jQuery里面把匹配元素集合称为matched elements set 或者
stack / 暂时还不明白为什么它叫做" pushStack ", stack在哪里.
300          *
301          * ANSWER:在链式的方法调用中，有很多的函数"return
this", 以便不用再写诸如"obj.method2;obj.method2()"这样的代码，而是直接
用
302          *
"obj.method1().method2()", 这样代码就会相当简洁。但是链式方法调用有一
个前提，那就要保证所有每一个"return this"的方法不能具有
303          * "破坏性".
就是函数说不能修改jQuery对象的匹配元素集合.如果某一个方法真的要修改匹
配元素集合，那么它就会调用pushStack方法,把当前的
304          * jQuery对象保存起来,以便以后使用end方法恢复这个jQuery对象.
305          */
306         pushStack: function( elems ) {
307             // Build a new jQuery matched element set
308             var ret = jQuery( elems );
```

```
309
310         // Add the old object onto the stack (as a reference)
311         ret.prevObject = this;
312
313         // Return the newly-formed element set
314         return ret;
315     },
316
317     // Force the current matched set of elements to become
318     // the specified array of elements (destroying the stack in the
    process)
319     // You should use pushStack() in order to do this, but maintain
    the stack
320     /*
321      *
    原文翻译:使当前匹配元素集合变成另一个特定的元素集合(在这个过程中原来的
    匹配元素的集合将会被破坏[也就是原来的那个集合不复存在].)
322      * 如果你想保持原来的那个匹配元素集合,那么你应该使用pushStack()
323      *
324      *
    使用这个函数将会把this对象原来的匹配对象集合重新设置成一个新的集合,而
    是使用pushStack()则保持原来的集合.
325      */
326     setArray: function( elems ) {
327         // Resetting the length to 0, then using the native Array push
328         // is a super-fast way to populate an object with array-like
    properties
329
330         //
    将length设为0,然后是用本地Array的push方法(也就是说将对象的指针传给Arra
    y.prototype.push,作为它的上下文,然后执行. this所代表的对象并不是
331         //
    Array对象,但是this又想使用Array对象才具有的push方法,于是使用了apply方
    法).
332         // 这是一个高速填充具有"类数组"属性的对象的方法
333         this.length = 0;
334
    //Array的push函数是在数组的末尾追加元素.值得注意的是Array类的push方法
    会修改this.length属性.这是
335
    //为什么选用Array的push的方法另外一个原因.如果不使用Array的push方法,那
    么就需要另外写代码来设置jQuery
336
    //对象的length属性,以使它的值与匹配元素集合中的元素个数相匹配.
337         Array.prototype.push.apply( this, elems );
338         //最后把追加了元素的jQuery对象返回
339         return this;
340     },
341
342     // Execute a callback for every element in the matched set.
343     // (You can seed the arguments with an array of args, but this is
344     // only used internally.)
345     /*
346      *
    原文翻译:为匹配元素集合内的每一个元素执行一遍回调函数.(你可以用数组的
    形式提供args这个参数, 不过它将被作为内部参数使用)
347      *
348      *
```

```
遍历匹配元素集合里面的每一个元素,并对每一元素调用callback函数进行处理.
 这样匹配元素集合还是原来那个集合,不过里面的元素都经过了callback的处理
349      * 可以看到遍历的操作实际上是调用了jQuery的静态方法each来完成的.
350      */
351     each: function( callback, args ) {
352
//this指的是一个jQuery对象(这个你应该很清楚).
由于一个jQuery对象是一个类数组的对象, 因此jQuery.each能像对待数组一样
353                                 //对待jQuery对象,
也因此jQuery匹配元素集合中的每一个元素都能被访问.
354         return jQuery.each( this, callback, args );
355     },
356
357     // Determine the position of an element within
358     // the matched set of elements
359     /**
360      * 上文翻译:确定一个元素在匹配元素集合中的位置(索引)
361      * @param {Object} elem
-要确定位置的那个元素.它可以是一个jQuery对象.
362      */
363     index: function( elem ) {
364         var ret = -1;
365
366         // Locate the position of the desired element
367         return jQuery.inArray(
368             // If it receives a jQuery object, the first element is
used
369             elem && elem.jquery ? elem[0] : elem
370         , this );
371     },
372
373     /**
374      *
获取或设置元素的属性值(这些属性可以是普通的属性值,如title,也可以是样式
属性值)
375      *
376      * name - 属性名称
377      * value - 属性值. 可以为空. 为空时表示要返回
378      * type -
表示要设置/获取的属性是一般元素属性,还是样式属性.如果没有传入这个参数,
则表明是一般的属性;如果有传入(如"curCSS"),则表示要设置样式属性
379      */
380     attr: function( name, value, type ) {
381         var options = name;
382
383         // Look for the case where we're accessing a style value
384         if ( name.constructor == String )
385             // 如果没有传入要设置的值,那么就是要获得该属性的值.
函数将返回匹配元素集合内首元素的相应属性
386             if ( value === undefined )
387                 return this[0] && jQuery[ type || "attr" ]( this[0],
name );
388
389                 /*
390                  * 我想解释一下上面那行return语句:
391                  * (1)
运算符&&的行为是这样的:对于它两边表达式,谁要把运算给中止了,就返回谁的
值.
```

```
392                     *
    在这里,this[0]如果是null或者是其他可以转换成false的值(如果0,
    undefined),
393                     * 那么运算中止,null(或其他与false等价 [ == ]
    的值)就会被返回. 若this[0]的确
394                     * 有值,那么运算不会中止, 继续&&右边的运算,
    并返回右边表达式计算的结果.
395                     *
396                     * (2) 而 " ||
    "运算符号也类似,左右两边谁把运算中止了, 就返回谁的值. 先计算左边
397                     * 的表达式, 如果不是false的等价值( ==
    false),就中止计算, 返回左边表达式的值.
398                     * 若左边表达返回的是一个和false等价的值,
    那么计算右边的表达式子, 并返回该表达式的
399                     * 值
400                     *
401                     * (3)
    如果给本函数传入了type,这个type一般就是"curCSS".
402                     */
403
404
405
406             //
    如果传入了要设置的值,让options成为一个"字典"(术语,即Key-Value对式的数
    据结构)
407             else {
408                 options = {};
409                 options[ name ] = value;
410             }
411
412         // Check to see if we're setting style values
413         //
    为每一个jQuery的匹配元素调用一个函数处理一下.这个函数处理的内容是:
414         //
    在当前的dom元素上,让opptions内记录的每一个属性都设置上相应的值.
415         // 最后将处理过的jQuery对象返回
416         return this.each(function(i){
417
418             /*
419              * 注意在这个函数内的 this, 它的含义已经不同.
    现在它指的是匹配元素集合中的每一个DOM元素
420              * 所有下面的代码中才会有this.style.
    如果this指向的是一个jQuery对象, 它又怎么会有这个
421              * 属性呢?
422              */
423
424             // Set all the styles
425             for ( name in options )
426                 jQuery.attr(
427                     type ?
    //如果有传入type,就表示要设置样式属性;如果没有则表示要设置一般的属性
428                         this.style :
429                         this,
430                     name, jQuery.prop( this, options[ name ], type, i
    , name )
431
    //调用prop取得正确的属性值(像"这个属性值是否要带单位?"这样的工作都交由
    prop来处理)
```

```
                );
            });
        },

        /*
         * 有两种情况:
         * (1)
当没有传入value值的时候,获取第一个匹配元素key所指定样式属性的值.
         * (2)
当有的传入value的时候,设置匹配元素集合中每一个元素上key所指定的样式属性值为value
         */
        css: function( key, value ) {
            // ignore negative width and height values
            // 原文翻译:在width 或者 height 上设置负值将会被忽略
            if ( (key == 'width' || key == 'height') && parseFloat(value) < 0 )
                value = undefined;

            // 最后调用attr函数获取样式值.注意,
如果没有传入最后一个参数,则表示获取/设置的是普通的属性(如title)而不是样式属性.
            return this.attr( key, value, "curCSS" );
        },

        /*
         * 返回/设置所有匹配元素的文本
         *
如果是返回文本的话,结果是由所有匹配元素包含的文本内容组合起来的文本。这个方法对HTML和XML文档都有效。
         * 如果是设置的话,则返回一个jQuery对象
         *
         * text - 要设置的文本内容(可选)
         */
        text: function( text ) {
            if ( typeof text != "object" && text != null )
                return this.empty().append( (this[0] && this[0].
ownerDocument || document).createTextNode( text ) );

            var ret = "";

            jQuery.each( text || this, function(){
                jQuery.each( this.childNodes, function(){
                    if ( this.nodeType != 8 )//8是comment节点
                        ret += this.nodeType != 1 ?
                            this.nodeValue :
                            jQuery.fn.text( [ this ] );
//递归获取this内的text,注意this是一个dom元素的引用
                });
            });

            return ret;
        },


        /* 以下为API文档摘抄:
         * 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。
480         *
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。
481         *
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。
482         */
483         wrapAll: function( html ) {
484             if ( this[0] )
485                 // The elements to wrap the target around
486                 jQuery( html, this[0].ownerDocument )
//jQuery(this[0].ownerDocument).find(html);
487                     .clone()
488                     .insertBefore( this[0] )
//这句之后,新clone出来的节点就在this[0]这个位置了
489                     .map(function(){
490 
491                         var elem = this;
492 
493                         while ( elem.firstChild )
494                             elem = elem.firstChild;
495 
496                         return elem;
497                     })
498                     .append(this);//this是一个dom 元素的引用
499 
500             return this;
501         },
502 
503         /*
504          * 把jQuery对象内的每一个匹配元素中的内容用指定的html包装起来
505          */
506         wrapInner: function( html ) {
507             return this.each(function(){
508                 jQuery( this ).contents().wrapAll( html );
509             });
510         },
511 
512 
513         /*
514          * jQuery对象内的每一个元素都用指定的html包装起来
515          */
516         wrap: function( html ) {
517             return this.each(function(){
518                 jQuery( this ).wrapAll( html );
519             });
520         },
521 
522         /**
523          * 向jQuery对象内的每个匹配的元素内部追加内容。
524          *
525          * 可以看到append实际上是调用了jQuery.fn.domManip来完成任务的.
526          * jQuery.fn.domManip其实是所有DOM修改方法(插入,删除等)的"母"方法.
527          *
只要在domManip方法的基础上修改调用参数,就能将domManip改头换面成另外一个
528          * 方法.
```

```
529          *
530          *
         本方法就是修改了domManip的最后一个参数callback,使之成为一个追加元素的
         方法.
531          *
532          * 同时参考jQuery.fn.domManip
533          */
534         append: function() {
535             /*
536              *
         domManip第二个参数的布尔(true)是说,所要进行的操作对象可能会是<table>,
         在IE中操作table有一些
537              * 限制,所以在此标出.具体是什么限制,可以参看domManip的注释
538              * 第二个布尔(false)是说arguments的参数是否需要翻转(即倒序).
539              */
540             return this.domManip(arguments, true, false, function(elem){
541                 if (this.nodeType == 1)
542                     this.appendChild( elem );
543             });
544         },
545 
546         /**
547          * 向jQuery对象内的每个匹配的元素内的子元素前前插入内容。
548          *
549          * 可以看到prepend实际上是调用了jQuery.fn.domManip来完成任务的.
550          * jQuery.fn.domManip其实是所有DOM修改方法(插入,删除等)的"母"方法.
551          *
         只要在domManip方法的基础上修改调用参数,就能将domManip改头换面成另外一
         个
552          * 方法.
553          *
554          *
         本方法就是修改了domManip的最后一个参数callback,使之成为一个追加元素的
         方法.
555          */
556         prepend: function() {
557             /*
558              *
         注意domManip的第三个参数(true),它表明arguments内的参数将会被倒序插入
559              *
         如要插入内容(即arguments)为'<div>1</div><div>2</div><div>3</div>',
560              *
         那么当他们被插入之后就会变成'<div>3</div><div>2</div><div>1</div>'
561              */
562             return this.domManip(arguments, true, true, function(elem){
563                 if (this.nodeType == 1)
564                     this.insertBefore( elem, this.firstChild );
565             });
566         },
567 
568         /*
569          * 在匹配元素集合内的每一个元素前插入内容
570          *
571          * 说明同上
572          */
573         before: function() {
574                             // 插入之前由domManip函数进行"把关",
         保证插入的内容是合法的(如<option>必须有<select>的包裹等)、IE
```

```
bug得到修复等等。
575             return this.domManip(arguments, false, false, function(elem){
576                 this.parentNode.insertBefore( elem, this );
577             });
578         },
579 
580         /*
581          * 在匹配元素集合内的每一个元素之后插入内容
582          */
583         after: function() {
584             /*
585              *
注意domManip的第三个参数(true),它表明arguments内的参数将会被倒序插入
586              *
如要插入内容(即arguments)为'<div>1</div><div>2</div><div>3</div>',
587              *
那么当他们被插入之后就会变成'<div>3</div><div>2</div><div>1</div>'
588              */
589                             // 插入之前由domManip函数进行"把关",
保证插入的内容是合法的(如<option>必须有<select>的包裹等)、IE
bug得到修复等等。
590             return this.domManip(arguments, false, true, function(elem){
591                 this.parentNode.insertBefore( elem, this.nextSibling );
592             });
593         },
594 
595         /*
596          * 将匹配的元素列表变为前一次的状态。
597          * 可以看到, 要保证这个需求能够实现,
jQuery其实是通过在jQuery对象内保存上一次操作的jQuery对象的引用来实现的
.这个对象就保存了上一次的匹配元素集合.
598          * 一般情况下, jQuery对象是没有prevObject这个属性的.
但是只要经过pushStack函数的操作之后,jQuery对象就具有了prevObject属性.
并且jQuery对象
599          *
变成了pushStack返回的那个新的jQuery对象.请查看jQuery.fn.pushStack
600          */
601         end: function() {
602             return this.prevObject || jQuery( [] );
603             // 以下是pushStack的代码,可以结合查看,以理解end的工作原理.
604             //  pushStack: function( elems ) {
605             //      var ret = jQuery( elems );
606             //      ret.prevObject = this;
607             //      return ret;
608             //  }
609         },
610 
611         /**
612          *
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。
613          *
可以看到它调用了jQuery.find函数完成任务.这是一个类静态方法而不是实例方
法.
614          *
另外它调用pushStack改变了jQuery对象匹配元素集合的内容,使用end函数能够
回到集合内容改变之前的状态.
615          * 具体可查看jQuery.fn.end,jQuery.fn.pushStack函数的注释.
```

```
616         * @param {string} selector - 用这个字符串指定需要选择的元素. 如'.titleBody','div', '#id'等
617         */
618        find: function( selector ) {
619
620 //查找匹配元素集合内每一个元素的后代元素(这些后代元素由selector指定),把这些后代元素全部集中起来放到elems中
621            var elems = jQuery.map(this, function(elem){
622                return jQuery.find( selector, elem );
623 //查找每一个元素的后代元素
624            });
625
626            //下面将利用pushStack将elems转化成一个新的jQuery对象,并将这个对象的preObject属性设为this, 最后pushStack将这个新对象返回,而find在接收到
627            // 这个新对象时后,又把它返回(return).
628
629
630            return this.pushStack( /[^+>] [^+>]/.test( selector ) || selector.indexOf("..") > -1 ?
631                jQuery.unique( elems ) :
632                elems );
633        },
634
635        /**
636         * 克隆一个jQuery对象.
637         *
638         * @param {Object} events
639         */
640        clone: function( events ) {
641
642            /*
643             * COMP:本来可以直接克隆的,但是IE浏览器的一些问题让我们不得不要针对它进行一些处理.
644             * 于是在map函数里面针对不同的浏览器进行的处理.具体是IE的什么问题,可以看看下
645             * 面那大段英文.
646             */
647            var ret = this.map(function(){
648 //注意jQuery对象的map方法返回的一个jQuery对象
649
650                //如果是IE浏览器
651                if ( jQuery.browser.msie && !jQuery.isXMLDoc(this) ) {
652                    // IE copies events bound via attachEvent when
653                    // using cloneNode. Calling detachEvent on the
654                    // clone will also remove the events from the orignal
655                    // In order to get around this, we use innerHTML.
656                    // Unfortunately, this means some modifications to
657                    // attributes in IE that are actually only stored
658                    // as properties will not be copied (such as the
659                    // the name attribute on an input).
660                    var clone = this.cloneNode(true),
 //参数true说明孩子节点也要一起被克隆
                        container = document.createElement("div");
                    container.appendChild(clone);
                    return jQuery.clean([container.innerHTML])[0];
```

```
661             }
662             else
663                 return this.cloneNode(true);
664         });
665 
666         // Need to set the expando to null on the cloned set if it
    exists
667         // removeData doesn't work here, IE removes it from the
    original as well
668         // this is primarily for IE but the data expando shouldn't
    be copied over in any browser
669         var clone = ret.find("*").andSelf().each(function(){
670             if ( this[ expando ] != undefined )
671                 this[ expando ] = null;
672         });
673 
674         // Copy the events from the original to the clone
675         // 把事件监听函数也一并拷贝到克隆的的对象上.
676         if ( events === true )
677             this.find("*").andSelf().each(function(i){
678                 if (this.nodeType == 3)//3是TextNode
679                     return;
680                 //获得this所指jQuery对象上的所有event集合
681                 var events = jQuery.data( this, "events" );
682 
683 
    //遍历这个集合上每一种事件类型.每一个type就是一个事件类型,如click等
684                 for ( var type in events )
685                     //遍历每一种事件类型上所有handler,即事件监听器.
    注意,不是"事件处理器".事件处理器
686                     //只有一个.
687                     for ( var handler in events[ type ] )
688                         //给clone[i]的type类型的事件添加上events[
    type ][ handler ]这个事件句柄(即事件监听器,叫句柄专业一些...).
689                         //更多的细节,请查看jQuery.event.add函数的细节.
690                         jQuery.event.add( clone[ i ], type, events[
    type ][ handler ], events[ type ][ handler ].data );
691             });
692 
693         // Return the cloned set
694         return ret;
695     },
696 
697     /**
698      * 在原来的匹配元素集合中去掉selector中指定的元素.
699      *
700      *
    由于改变了匹配元素集合中的元素,所以调用pushStack来设置一个"回退点".如
    果调用jQuery.fn.end函数就能恢
701      * 复为原来的那个jQuery对象.
702      *
703      * @param {Object} selector
    它的取值范围跟jQuery.fn.init的selector一样.
704      */
705     filter: function( selector ) {
706         return this.pushStack(
707             /*
708              *
```

```
如果selector是一个函数,那么使用这个函数对jQuery对象内匹配元素集合中的每一个元素进行过滤
709              */
710             jQuery.isFunction( selector ) &&
711             jQuery.grep(this, function(elem, i){
712                 return selector.call( elem, i );
713             }) ||
714
715             /*
716              * 如果不是函数, 那就交给jQuery.multiFilter进行处理.
717              * this 指明了一个过滤的上下文,
即要在这个范围内(它的匹配元素集合内)进行过滤.
718              *
multiFilter在过滤匹配元素集合方面处于核心的地位,许多方法都是用它来完成任务的.
719              */
720             jQuery.multiFilter( selector, this ) );
721     },
722
723     /**
724      * 在jQuery对象的匹配元素集合中去掉由seletor指定的元素.
725      *
726      * @param {Object} selector selector的取值范围跟jQuery.fn.init一样
727      */
728     not: function( selector ) {
729
730         //要将selector分开两种情况来讨论哦:字符串还是数组类结构?
731
732         //如果是字符串
733         if ( selector.constructor == String )
734             // test special case where just one selector is passed in
735             // isSimple是一个简单的正则选择器,如 '#id'
这样的选择器,它没有空格' '.
736             // 像 '.div1 h1'这样的选择器就算是一个较为复杂的选择器
737             if ( isSimple.test( selector ) )
738                 // multiFilter的最后一个参数 true
表示启用'非模式'.具体请看jQuery.multiFilter的注释
739                 //
'非模式'如果启动了,selector中指定的元素就不要.如果不启动,则selector中指定的元素就是结果集.
740                 return this.pushStack( jQuery.multiFilter( selector,
this, true ) );
741             else
742                 selector = jQuery.multiFilter( selector, this );
743
744         //看看selector是不是类数组对象
745         var isArrayLike = selector.length && selector[selector.length
 - 1] !== undefined && !selector.nodeType;
746
747         return this.filter(function() {
748             //注意, this指的是匹配元素集合中的每一个元素
749             return isArrayLike ?
750
//selector如果是数组,那么这个数据就划定了一个范围.若this所指的元素不在这个范围内
751                 //就把元素保留(这时jQuery.inArray(this,selector)
< 0 将返回true).
752                 jQuery.inArray( this, selector ) < 0 :
```

```
753
754
//如果selector不是类数组的元素,那么只要不跟selector在逻辑上相等,就可以保留
755                     this != selector;
756         });
757     },
758
759     /**
760      * 将selector指定的元素添加到匹配元素集合中去.
761      *
762      *
由于修改了匹配元素集合的内容,所以使用了pushStack.具体参见pushStack以及end的注释
763      * @param {Object} selector
764      */
765     add: function( selector ) {
766         return this.pushStack( jQuery.unique( jQuery.merge(
767             this.get(),
768             typeof selector == 'string' ?
769                 jQuery( selector ) :
770                 jQuery.makeArray( selector )
771         )));
772     },
773     /**
774      *
返回一个布尔值,确定匹配元素集合中的元素是否在selector指定的一个范围之内.
775      * 如果有一个在,那么就返回true;
776      * 如果没有一个在, 或者selector是一个无效的选择器,返回false.
777      *
778      * @param {Object} selector 任意合法的selector.参见jQuery.fn.init
779      */
780     is: function( selector ) {
781         /*
!运算符会先把它的运算数转换成一个布尔类型的值.任何值x,两次取反(!!)之后都可以把它转换成为一个布尔值
782          *
multiFilter是一个'多功能'过滤器,可以用它来过滤掉不需要的元素(给它的三个参数传入true);也可以用它来
783          *
过滤剩下selector所指定的元素(第三个参数为false或者不传入第三个参数).
784          */
785         return !!selector && jQuery.multiFilter( selector, this ).
length > 0;
786     },
787
788     /**
789      *
返回一个布尔,确定匹配元素集合中的元素是否具有selector指定的类名.
790      * 规则同 is 函数
791      * @param {Object} selector
792      */
793     hasClass: function( selector ) {
794         return this.is( "." + selector );
795     },
796
797     /**
```

```
798         * 设置每一个匹配元素的值。或者是获取匹配元素集合中首元素的值
799         * 如果有一个非空的返回值,说明设置成功.
800         *
801         * @param {Object} value
802         */
803        val: function( value ) {
804            //如果value为undefined说明要取值而不是要设值.
805            if ( value == undefined ) {
806
807                //如果匹配元素集合不是空的
808                if ( this.length ) {
809                    var elem = this[0];
810
811                    // We need to handle select boxes special
812                    // 如果节点是<select>,那么需要特别的处理
813                    if ( jQuery.nodeName( elem, "select" ) ) {
//nodeName函数测试一个节点不是指定类型的节点
814                        var index = elem.selectedIndex,
815                            values = [],
816                            options = elem.options,
817                            one = elem.type == "select-one";
//这个<select>是多选的还是单选的?
818
819                        // Nothing was selected
820                        if ( index < 0 )
821                            return null;
822
823                        // Loop through all the selected options
824                        /*
825                         * 这个for循环的初始化部分使用了嵌套的' ? :
'运算符,比较晦涩.
826                         */
827                        for ( var i = one ? index : 0, max = one ? index
+ 1 : options.length; i < max; i++ ) {
828                            var option = options[ i ];
829
830                            if ( option.selected ) {
831                                // Get the specifc value for the option
832                                /*
833                                 *
COMP:IE的option获取值的方式竟然是这么地麻烦.各位记住就好.
834                                 */
835                                value = jQuery.browser.msie && !option.
attributes.value.specified ? option.text : option.value;
836
837                                // We don't need an array for one selects
838                                // 如果是单选的<select>,就把值返回
839                                if ( one )
840                                    return value;
841
842                                // Multi-Selects return an array
843                                // 如果是多选的,就把值放进一个数组.
844                                values.push( value );
845                            }
846                        }
847
848                        return values;
849
```

```
                // Everything else, we just grab the value
                }//如果不是<select>而是其他的节点
                else
                    // /r 匹配一个回车符
                    return (this[0].value || "").replace(/\r/g, "");

            }

            return undefined;
        }

        if( value.constructor == Number )
            value += '';

        return this.each(function(){

            /*
注意哈,this现在不是指向jQuery对象,而是匹配元素集合内的每一个元素 */

            if ( this.nodeType != 1 /* ELEMENT_NODE */)
                return;
            /*
             *
如果要设置的值是一个数组,并且each当前遍历到的元素的是radio或者checkbox
的
             * 那么就设置他们的checked值.设置的规则如下:
             *
如果当前元素的值(this.value)或者名字(this.name)在数组所划定的范围之内,
             * 就把checked值设为true.否则为false.
             */
            if ( value.constructor == Array && /radio|checkbox/.test(
 this.type ) )
                this.checked = (jQuery.inArray(this.value, value) >=
0 ||
                    jQuery.inArray(this.name, value) >= 0);

            // 如果当前元素是<select>
            else if ( jQuery.nodeName( this, "select" ) ) {
                var values = jQuery.makeArray(value);
                //
由当前这个<select>元素的所有option孩子新建一个jQuery对象.这样就能够利
用jQuery的方法
                // 方便地操作这些孩子了.
                jQuery( "option", this ).each(function(){
//用each来遍历<select>的每一个<option>孩子

                    /* 注意啦,现在 this 指向的是一个<option>元素咯 */

                    /*
                     *
如果孩子的value(这个孩子是<option>元素来的)或者text
在values所界定的范围之内
                     * 那么就把孩子的selected设置为true.
                     */
                    this.selected = (jQuery.inArray( this.value,
values ) >= 0 ||
                        jQuery.inArray( this.text, values ) >= 0);
                });
```

```
896
897                 //如果values的长度为0,就设置selectedIndex为-1.
898                 if ( !values.length )
899                     this.selectedIndex = -1;
900
901             }
902             // 都不是以上情况,那就直接把值设置进来就是了
903             else
904                 this.value = value;
905         });//end function 'each'
906     },
907
908     /**
909      * 设置/获取元素的html内容
910      * 在value没有指定的时候返回匹配元素集合中首元素的innerHTML.
911      *
    如果指定了value,那么匹配元素集合中每一个元素的子元素为由value生成的元
    素.
912      *
913      * 同时参考:jQuery.fn.append
914      *
915      * @param {Object} value
916      */
917     html: function( value ) {
918         return value == undefined ?
919             (this[0] ?
920                 this[0].innerHTML :
921                 null
922             )
923                 :
924             this.empty().append( value );
925     },
926     /**
927      *
    在匹配元素集合中的每一个元素之后插入value做为兄弟节点,并调用remove方法
    将value值内的类似'<>'的符号
928      * 去掉.
929      *
930      * @param {Object} value
931      */
932     replaceWith: function( value ) {
933         return this.after( value ).remove();
934     },
935
936     /**
937      *
    将匹配的元素集合缩减为一个元素。这个元素在匹配元素集合中的位置变为0,
    而集合长度变成1。
938      * 最后用这个集合重新构建一个jQuery对象,并将其返回.
    由于修改了匹配元素集合,所以在slice方法体内使用了
939      * pushStack来保留一个'恢复点',
    以便能使用jQuery.fn.end方法恢复到以前的状态.
940      *
941      * 同时参考jQuery.fn.pushStack 和 jQuery.fn.slice
942      *
943      * @param {Object} i i指示要将哪一个保留元素
944      */
945     eq: function( i ) {
```

```
946             return this.slice( i, i + 1 );
947         },
948
949         /**
950          * 将匹配的元素集合缩减为若干个元素。
951          * 最后用这个集合重新构建一个jQuery对象,并将其返回.
    由于修改了匹配元素集合,所有使用pushStack
952          * 来保留一个'恢复点',
    以便能使用jQuery.fn.end方法恢复到以前的状态.
953          *
954          * 同时参考jQuery.fn.pushStack
955          *
956          * @param 传入的参数需要符合JavaScript
    Core中Array对象的slice方法的要求指示要将哪一个保留元素
957          * slice需要两个参数:
958          * 第一个参数指定截取的位置,第二个参数指定截取长度.
959          */
960         slice: function() {
961             return this.pushStack( Array.prototype.slice.apply( this,
    arguments ) );
962         },
963
964         /**
965          *
    使用callback处理匹配元素集合中的每一个元素,完后用pushStack新建一个jQue
    ry对象,最后返回这个新
966          * 的jQuery对象.
967          *
968          * 具体细节可以参看pushStack的注释.
969          *
970          * 注意:
971          * 本实例方法调用了jQuery.map静态函数完成任务,它返回值是一个数组.
972          * 而jQeury.fn.map也就是本方法是一个实例方法,
    它的返回值却是一个jQuery的对象.
973          *
974          * @param {Function} callback 用来做映射的函数.
975          */
976         map: function( callback ) {
977             return this.pushStack( jQuery.map(this, function(elem, i){
978
    //callback将会作为elem的方法来调用,
    那么callback代码内的this指的就是正在处理的匹配元素集合中的元素.
979                 return callback.call( elem, i, elem );
980             }));
981         },
982
983         /**
984          * 将this.prevObejct内的匹配元素集合也加进当前的匹配元素集合
985          * 注意:this.prevObject是一个jQuery对象的引用.
986          */
987         andSelf: function() {
988             return this.add( this.prevObject );
989         },
990
991
992         data: function( key, value ){
993             var parts = key.split(".");
994             parts[1] = parts[1] ? "." + parts[1] : "";
```

```
995
996         if ( value === undefined ) {
997             var data = this.triggerHandler("getData" + parts[1] + "!", [parts[0]]);
998
999             if ( data === undefined && this.length )
1000                 data = jQuery.data( this[0], key );
1001
1002             return data === undefined && parts[1] ?
1003                 this.data( parts[0] ) :
1004                 data;
1005         } else
1006             return this.trigger("setData" + parts[1] + "!", [parts[0], value]).each(function(){
1007                 jQuery.data( this, key, value );
1008             });
1009     },
1010
1011     removeData: function( key ){
1012         return this.each(function(){
1013             jQuery.removeData( this, key );
1014         });
1015     },
1016
1017
1018     /* domManip 其实是 Dom manipulate的缩写.
1019      *
让每一个jQuery匹配元素集合内的元素都执行一遍callbak(callback可以是插入、修改等), args为参数.
1020      *
与此同时本函数会对args进行处理以保证args在dom内的正确性(如<option>必须要有<select>的包裹等)
1021      *
另外本函数也保证了包含在args内的脚本能在具体的dom结构生成之后才执行,避免了出错.
1022      */
1023     /*
1024      * 为了能够更好地理解domManip函数的功能,
可以结合jQuery.fn.append来说明此函数的作用:
1025      * 在append函数中有这样的代码:return this.domManip(arguments, true, fals, function(elem){
1026
             if (this.nodeType == 1)
1027
             this.appendChild( elem );
1028
         });
1029      *
1030      */                     //true  //false  //function(){...
1031     domManip: function( args, table, reverse, callback ) {
1032         // args是类似这样的字符串:"<b>Hello</b>"
1033
1034         // 如果length>1 就要clone
1035         var clone = this.length > 1, elems;
1036
1037         //
遍历jQuery对象(this)匹配元素集合中的每一个元素,并调用匿名函数对集合内每一个元素进行处理
```

```
1038            // 处理的内容是:为每一个元素进行callback操作,参数是args
1039            // 例如:
        如果callback的功能是为元素追加内容(即append),则args就是要追加的具体内
        容
1040            return this.each(function(){
1041                if ( !elems ) {
1042                    elems = jQuery.clean( args, this );
        //jQuery.clean之后,elems变成一个数组.这个数组内装的是一些dom元素.这些d
        om元素由args内表示XHTML的
1043
        //字符串变成清空内容的dom元素得来.具体请查看jQuery.clean函数.
1044
1045                    if ( reverse )
1046                        elems.reverse();
1047                }
1048
1049                var obj = this;
        //为了方便叙述下面的代码分析,这里作一个标记[1]. 注意,
        this是一个dom元素的引用. 不是jQuery对象的引用.
1050
1051                //在IE中,如果需要在table中操作tr(如插入一个tr),
        IE要求你在tbody内进行操作.
1052                // 以下的if就是说要在tbody内进行操作,
        如果没有tbody就自己建一个,然后再操作
1053                if ( table && jQuery.nodeName( this, "table" ) && jQuery.
        nodeName( elems[0], "tr" ) )
1054
        //appendChild返回值是一个指向新增子节点的引用
1055                    obj = this.getElementsByTagName("tbody")[0] || this.
        appendChild( this.ownerDocument.createElement("tbody") );
1056
1057
1058                // 如果发现操作(如插入操作)的内容中含有脚本,
        先把这些脚本装进这个集合,最后才来运行他们
1059                var scripts = jQuery( [] );
1060
1061                // 对于每一个要操作(如插入操作)的对象(elem)进行一些处理.
1062                // 这些处理的内容是: 如果对象里面含有脚本,
        将这些脚本添加到一个脚本集合内,留待后面执行.
1063                jQuery.each(elems, function(){
1064                    var elem = clone ?          //true的意思是说,
        把绑定在jQuery(this)上的事件也一并克隆
1065                        jQuery( this ).clone( true )[0] ://
        clone返回的是一个jQuery对象,这对象的匹配元素集合内
1066                                                        //
        只有一个元素, 所以clone(true)[0]就是this的一个副本
1067                        this;
1068
1069                    // execute all scripts after the elements have been
        injected
1070                    //
        如果插入的内容里面含有script,用集合先把这些script装起来,
        等到最后所有的内容都插入完毕了, 执行这个集合里面的所有脚本
1071                    if ( jQuery.nodeName( elem, "script" ) )
1072                        scripts = scripts.add( elem );
1073                    else {
1074                        // Remove any inner scripts for later evaluation
1075                        if ( elem.nodeType == 1 )
```

```
1076                            scripts = scripts.add( jQuery( "script", elem ).remove() );
1077
1078                        // Inject the elements into the document
1079                        // 这里的obj要么是上面代码[1]处的this( obj = this ), 要么是this内的tbody( obj = this.getElementByTagName("tbody")||... )
1080                        // 最后在obj上执行callback指定的操作类型(elem为参数).
1081                        // 例如, callback是一个追加内容的函数, 那么这里就让obj追加内容elem
1082                        callback.call( obj, elem );
1083
1084                        /*
1085                         * 对于上面的这行代码还有一点要补充:
1086                         * 其他函数在使用domManip都是像这样使用的:
1087                         * domManip(arguments,true,true,function(elem){
1088                         *      // callback body
1089                         * });
1090                         * 请特别注意callback body 内的this和参数elem.
1091                         * 由于callback.call(obj,elem)使用了obj作为函数调用上下文,故obj自然就成为了this
1092                         * 所指向的对象.又于是,给domManip传入的callback只有一个形式参数elem.
1093                         *
1094                         */
1095
1096                    }
1097                });//end each
1098
1099                //为一个元素插入完内容之后(即注入完HTML),就个执行刚才保存的脚本
1100                scripts.each( evalScript );
1101            });//end each
1102        }
1103    };
1104
1105
1106
1107    // Give the init function the jQuery prototype for later instantiation
1108    // 通过这一句之后, jQuery.fn.init也能实例化一个jQuery对象的对象了.
1109    jQuery.fn.init.prototype = jQuery.fn;
1110
1111    function evalScript( i, elem ) {
1112        if ( elem.src )
1113            jQuery.ajax({
1114                url: elem.src,
1115                async: false,
1116                dataType: "script"
1117            });
1118
1119        else
1120            jQuery.globalEval( elem.text || elem.textContent || elem.innerHTML || "" );
1121
1122        if ( elem.parentNode )
1123            elem.parentNode.removeChild( elem );
```

第二次实例化

```
1124    }
1125
1126    function now(){
1127        return +new Date;
1128    }
1129
1130    /**
1131     *
        这是jQeury核心中很重要的一个函数.通过它我们就可以轻松地在jQuery或者jQu
        ery对象中随意扩展自己想要的方法
1132     */
1133    jQuery.extend = jQuery.fn.extend = function() {
1134        // copy reference to target object
1135        var target = arguments[0] || {}, //target是被扩展的对象,
        默认是第一个参数(下标为0)或者是一个空对象{}
1136            i = 1,
        //i是一个"指针",它指向扩展对象.也就是说要把这个对象的属性或方法扩展到
        被扩展对象上
1137            length = arguments.length, //参数的长度.
        通过这个长度来判断扩展的模式
1138            deep = false, //是否要进行深度扩展(拷贝).
        当一些属性是一个对象,对象内又有对象时,就需要取舍了:到底要不是拷贝整个
        对象树?
1139            options;//当前正在拷贝的扩展对象的引用
1140
1141        // Handle a deep copy situation
1142        //
        如果传进来的首个参数是一个boolean类型的变量，那就证明要进行深度拷贝。
        而这时传进来的argumens[1]就是要拷贝的对象.如果是这种情况,那就要做一些
1143        // "矫正"工作,
        因为这个时候,target变量指向的是一个布尔变量而不是我们要拷贝的对象.
1144        if ( target.constructor == Boolean ) {
1145            deep = target;//保存target的布尔值
1146            target = arguments[1] || {};
        //让target真正指向我们要拷贝的对象.
1147
1148            // skip the boolean and the target
1149            // 可以说是一个指针，现在指向argument[2].
        arguments[0],arguments[1]已经得到处理,
        剩下需要处理的就是arguments[2],arguments[3],...
1150            // 后面的参数.
1151            i = 2;
1152        }
1153
1154        // Handle case when target is a string or something (possible in
        deep copy)
1155        // 如果target不是objet 并且也不是function 就默认设置它为{};
1156        if ( typeof target != "object" && typeof target != "function" )
1157            target = {};
1158
1159        // extend jQuery itself if only one argument is passed
1160        // 翻译:如果只传入了一个参数, 那么扩展的就是jQuery自身:
1161        // 如果使用jQuery.extend来扩展，那么this 就是jQuery.
        这样的话，参数中的函数就会作为jQuery下的静态方法。
1162        // 如果使用jQuery.fn.extend来扩展， this
        指的就是jQuery.fn了。参数中所的函数或者属性就会作为jQuery对象的方法或
        属性了。
1163        if ( length == i ) {
```

```
1164            target = this;
1165            --i;
1166        }
1167
1168        for ( ; i < length; i++ )
1169            // Only deal with non-null/undefined values
1170            // 只有那些非null的扩展对象才把它扩展到被扩展对象上来.
1171            if ( (options = arguments[ i ]) != null )
1172                // Extend the base object
1173                // 扩展被扩展对象(base
    object),将options内的属性或方法扩展到被扩展对象中来
1174                for ( var name in options ) {
    //现在要遍历每一个加进来的方法或属性
1175                                    //target是被扩展对象
    //options是扩展对象, 它的方法或属性将会被扩展到target上
1176                    var src = target[ name ], copy = options[ name ];
1177
1178                    // Prevent never-ending loop
1179                    // taget == copy
    说明要加进来的引用是指向自己的，这在要进行深度拷贝时就糟糕了。所以碰到
    这样的情况就跳过，不把自己的引用作为自己的一个成员
1180                    if ( target === copy )
1181                        continue;
1182
1183                    // Recurse if we're merging object values
1184                    //
    使用递归的方法实现深度拷贝                                    //
    这个条件是说,要是没有nodeType, 就是非Dom对象引用, 可以对它进行深度拷贝
1185                    if ( deep && copy && typeof copy == "object" && !copy
    .nodeType )
1186
1187                        target[ name ] = jQuery.extend( deep,
1188                            // Never move original objects, clone them
1189                            src || ( copy.length != null ? [ ] : { } )
1190                        , copy );
1191
1192                    // Don't bring in undefined values
1193                    // 如果要加进来的引用不是对象的引用(
    可能是函数、简单变量，只要不是undefined ) 那就把引用加进来:
    可能是覆盖也可能是新建name这个属性或方法
1194                    else if ( copy !== undefined )
1195                        target[ name ] = copy;
1196
1197                }
1198
1199        // Return the modified object
1200        // 把扩展好的对象返回
1201        return target;
1202    };
1203
1204              /*
    定义好了jQuery.extend和jQuery.fn.extend方法之后,以后我们就使用这个方法
    来为jQuery
1205               * 或者jQuery.fn,又或者jQuery.fx等随意添加(扩展)方法了.
1206               */
1207
1208
1209    var expando = "jQuery" + now(),
```

```
//当元素需要缓存数据时,这样使用expando: id = elem[expando];data = jQueyr.catche[id];
1210        uuid = 0, //如果一个元素需要缓存数据,那么uuid++就会成为它的缓存区全局唯一编号.
1211        windowData = {}, //这是数据缓存区.
1212
1213        // exclude the following css properties to add px
1214        // 以下这些CSS属性是不需要加单位'px'的
1215        exclude = /z-?index|font-?weight|opacity|zoom|line-?height/i,
1216        // cache defaultView
1217        //在这里定义一个defaultView在需要的函数里就可以直接调用,而不用再写一长串document.XXXXXXXX了.
1218        defaultView = document.defaultView || {};
1219
1220
1221
1222
1223
1224    // ----------------------------------- jQuery静态核心函数 ----------------------------------------
1225    //这些静态函数为jQuery对象实例方法或者其他需要的函数所调用.jQuery很多的实例方法实际上是调用了这里定义的方法'幕后'完成任务的
1226    //
1227    //通过jQuery.extand这个'巨大'的函数调用,许多jQuery的核心函数都被定义并加进了jQuery这个命名空间里.
1228    //注意,这些函数都是静态的.这意味着你调用他们的时候必须使用完整的限定符号,如果jQuery.noConfiict,并且不能像jQuery实例方
1229    //  法那样调用它,如jQuery(seloector).swap是不合法的.
1230    //-------------------------------------------------------------------------------
1231    jQuery.extend({
1232        /**
1233         * 将命名空间$归还.调用这个函数就不能再用jQuery或者$了
1234         * @param {Object} deep
1235         */
1236        noConflict: function( deep ) {
1237            window.$ = _$;//刚才是存起来的$引用现在'还'给人家.
1238
1239            if ( deep )
1240    //如果传入deep,说明连jQuery这个关键字的使用权也要放弃.真的比较'deep'了.
1241                window.jQuery = _jQuery;
1242
1243            return jQuery;
1244        },
1245
1246        // See test/unit/core.js for details concerning this function.
1247        /**
1248         * 测试一个对象是不是Function.
1249         * 可以看到,检查一个对象是不是函数还是比较多'工序'的.
1250         * @param {Object} fn
1251         */
1252        isFunction: function( fn ) {
```

```
1253            /*
1254             * 这里使用了将函数转化为字符串(使用 函数对象+""
    的方法),然后用正则表达式测试是否符合一个函数所应该有的模式.
1255             * 所以下情况会造成混淆:
1256             * (1) 一个内容为'function(){}'的字符串;
1257             * (2)
    一个含有混淆字符串元素的数组,如['function','()','{}'],这个数组一转化成
    字符串,就成为了(1)所说的情况
1258             *
1259             * 至于为什么不用typeof fn
    来确定一个元素是否为function,这点可能是'兴趣爱好'问题,又可能是由于某些
    老式浏览器
1260             *
    会返回不正确的结果(我在IE5.5+,Firefox3,Safari3.0.4,Chrome,Opera9.62上
    都测试过typeof,结果正常 ).请
1261             * 读者自行斟酌.
1262             */
1263
1264            return !!fn && typeof fn != "string" && !fn.nodeName &&
1265                fn.constructor != Array && /^[\s[]?function/.test( fn +
    "" );
1266        },
1267
1268
1269        // check if an element is in a (or is an) XML document
1270        /**
1271         * 检查一个元素是否是XML的document
1272         * @param {Object} elem
1273         */
1274        isXMLDoc: function( elem ) {
1275
    //body是HTMLDocument特有的节点常用这个节点来判断当前的document是不是一
    个XML的文档引用
1276
    //注意,HTMLDocment接口是XMLDocument的扩展,即HTMLDocument中特定于处理HT
    ML文档的方法
1277            //(如getElementById等)是不能用在XML文档使用的.
1278            return elem.documentElement && !elem.body ||
1279                elem.tagName && elem.ownerDocument && !elem.ownerDocument
    .body;
1280        },
1281
1282        // Evalulates a script in a global context
1283        /**
1284         * 原文翻译:
1285         * 在全局的作用域中运行脚本
1286         *
1287         * @param {Object} data
1288         */
1289        globalEval: function( data ) {
1290            //调用trim函数将data两头的空格去掉.
1291            data = jQuery.trim( data );
1292            //如果是有脚本内容的就运行,没有就结束函数,返回.
1293            if ( data ) {
1294                // Inspired by code by Andrea Giammarchi
1295                //
    http://webreflection.blogspot.com/2007/08/global-scope-evaluation-and-
    dom.html
```

```
1296            var head = document.getElementsByTagName("head")[0] ||
    document.documentElement,
1297                script = document.createElement("script");
1298
1299            script.type = "text/javascript";
1300            if ( jQuery.browser.msie )
1301                script.text = data;
1302            else
1303                script.appendChild( document.createTextNode( data ) );
1304
1305            // Use insertBefore instead of appendChild  to
    circumvent an IE6 bug.
1306            // This arises when a base node is used (#2709).
1307            head.insertBefore( script, head.firstChild );
1308            head.removeChild( script );
1309        }
1310    },
1311
1312    /*
1313     * 判断一个元素的nodeName是不是给定的name
1314     *
1315     * elem - 要判定的元素
1316     * name - 看看elem.nodeName是不是这个name
1317     */
1318    nodeName: function( elem, name ) {
1319        return elem.nodeName && elem.nodeName.toUpperCase() == name.
    toUpperCase();
1320    },
1321    /**
1322     *
    全局数据缓存区.每一个需要缓存数据元素都会在这里开辟一个空间存自己的数
    据
1323     */
1324    cache: {},
1325
1326    /**
1327     * 在jQuery全局数据缓存区中缓存数据.
1328     *
1329     * 注意别被"数据缓存区"吓到了.在技术上它不过就是一个对象,存了一些
1330     * 数据而已.
1331     *
1332     * @param {Object} elem 要在这个元素上存放数据
1333     * @param {Object} name 数据的键名
1334     * @param {Object} data 数据的键值
1335     */
1336    data: function( elem, name, data ) {
1337        elem = elem == window ?
1338            windowData :
1339            elem;
1340
1341                            /*
    获取元素的id.这个id被存在一个叫expando的属性里.
1342                             * expando 只是一个由 " jQuery " + now()
    组成的字符串
1343                             *
    它将作为elem的一个属性,同时这个属性的值也是全局数据缓存区中
1344                             *
    某一块的名字.根据这个名字就可以找到元素相应的缓存数据.
```

```
1345                            *
    注意别被"数据缓存区"吓到了.在技术上它不过就是一个对象,存了一些
1346                            * 数据而已.
1347                            */
1348            var id = elem[ expando ];
1349
1350            // Compute a unique ID for the element
1351            //如果元素还没有expando编号,给它新建一个
1352            if ( !id )
1353                id = elem[ expando ] = ++uuid;
1354
1355            // Only generate the data cache if we're
1356            // trying to access or manipulate it
1357            /*
1358             *
    如果有传入name属性,那么就在jQuery.cache区内新建一个属于本元素的cache
1359             */
1360            if ( name && !jQuery.cache[ id ] )
1361                jQuery.cache[ id ] = {};
1362
1363            // Prevent overriding the named cache with undefined values
1364            // 翻译(意译): 别让未定义的值设置进来.
1365            if ( data !== undefined )
1366                jQuery.cache[ id ][ name ] = data;
1367
1368            // Return the named cache data, or the ID for the element
1369            //
    最后把缓存的数据返回.值得注意的是,在没有传入name的情况之下,函数返回元
    素的id.这种不给data函数传name值的用法
1370            // 可以参见jQuery.unique函数或者jQuery.find函数
1371            return name ?
1372                jQuery.cache[ id ][ name ] :
1373                id;
1374        },
1375
1376        /**
1377         * 取消元素的缓存的数据.
1378         *
1379         * @param {Object} elem
1380         * @param {Object} name
1381         */
1382        removeData: function( elem, name ) {
1383            elem = elem == window ?
1384                windowData :
1385                elem;
1386            //取得元素的全局ID
1387            var id = elem[ expando ];
1388
1389            // If we want to remove a specific section of the element's
    data
1390            // 下面就删除这些数据
1391            if ( name ) {
1392                if ( jQuery.cache[ id ] ) {
1393                    // Remove the section of cache data
1394                    delete jQuery.cache[ id ][ name ];
1395
1396                    // If we've removed all the data, remove the
    element's cache
```

```
1397                    name = "";
1398
1399                    /*
以下这个for循环不断地从jQuery.cache[id]取属性名出来,并放入到name,
1400                     *
一旦name获得了一个可以转化为true的值,for的循环体就会被执行.执行之后
1401                     *
居然是break...于是就起到了一个作用:检测元素是否还有自定义的属性.
1402                     * 注意,元素的继承属性不能被for
in循环枚举.例如从Object中继承下来的
1403                     * toString函数就不能被for in访问到.
1404                     */
1405                    for ( name in jQuery.cache[ id ] )
1406                        break;
1407
1408                    if ( !name )
//如果如果name仍然是空的,那么就说明jQuery,cache[id]中再也没有数据了,可
以把这个
1409
//数据缓存去删掉了.递归调用removeData.这时,removeData函数执行的就是下
面那个
1410                                //else里面的代码了.
1411                        jQuery.removeData( elem );
1412                }
1413
1414            // Otherwise, we want to remove all of the element's data
1415            }
1416
1417            // Otherwise, we want to remove all of the element's data
1418            //
有些人英语比较水,我翻译一下,呵呵:否则,我需要删除元素所有缓存的数据.
1419            //
也就是说,当没有把name值传进来的时候(但有传一个elem进来),说明要删除元素
elem上的所有缓存数据.
1420            else {
1421                // Clean up the element expando
1422                /*
删除元素上的expando属性.先用delete操作符删除这个元素的expanddo属性.如
果出问题了,那问题可能是
1423                 * 由当前浏览器是IE所致,在catch中尝试removeAttribute.
1424                 *
1425                 * TEACH教学时间:
1426                 *
removeAttribute方法属于w3c标准中1级DOM的API.而1级DOM在现代浏览器中已经
得到了相当广泛的支持.
1427                 *
也就是说,如果不考虑老式的浏览器,直接使用removeAttribute而不用try/catch
也是可以的.delete操作
1428                 *
符号属于JavaScript核心,也就是说,只要是个JavaScript的解释器就不应该不认
识delete.因此下面的
1429                 * try/catch先用delete试试.不行了就用removeAttribute.
1430                 */
1431                try {
1432                    /* TEACH教学时间:
1433                     *
注意啦,delete操作符号只是删除元素的属性而已,它并不会删除属性所引用的内
存地址的内容.回收内存空间那
```

```
1434                     *
是垃圾回收机制的事情.程序员唯一要能做的就是相信它能够起作用!这跟C++的delete相当不同,C++程序员要注意.
1435                     *
另外,delete可以删除一个属性,但是不能删除由var定义的变量.不过隐式定义变量可以删除.
1436                     */
1437                    delete elem[ expando ];
1438                } catch(e){
1439                    // COMP: 以下是IE在delete方面的一个bug:
1440                    // IE has trouble directly removing the expando
1441                    // but it's ok with using removeAttribute
1442                    //
翻译:直接删除exando属性在IE中会出问题.不过用removeAttribute一样能达到删除属性的目的.
1443                    if ( elem.removeAttribute )
1444                        elem.removeAttribute( expando );
1445                }
1446
1447                // Completely remove the data cache
1448                //
完全删除掉数据.跟上面的忠告一样,delete只是删除一个地址的引用,要回收那一块内存空间那必须得垃圾机制说了算.
1449                delete jQuery.cache[ id ];
1450            }
1451        },
1452
1453        // args is for internal usage only
1454        /**
1455         *
遍历集合中的每一个元素.对每一个元素调用callback对其进行处理.jQuery代码中大量使用到这个函数.
1456         *
1457         * object -
就是要遍历的数组或者类数组对象,或者干脆就是一个普通对象
1458         * args -
1459         */
1460        /**
1461         *
1462         * @param {Object} object -
就是要遍历的数组或者类数组对象,或者干脆就是一个普通对象
1463         * @param {Object} callback - 遍历object时, 执行的处理函数.
1464         * @param {Object} args - 是要给callback使用的内部参数
1465         */
1466        each: function( object, callback, args ) {
1467            var name, i = 0, length = object.length;
1468
1469            // 如果 给callback传进了args
1470            if ( args ) {
1471                // length == undefined 说明 object不是类数组( array-like )对象,那就不能使用下标来访问了所以只能 for ( name in object )
1472                if ( length == undefined ) {
1473                    for ( name in object )//遍历
object内的每一个属性,并将属性作为callback的上下文,args作为callback的参数来调用callback
1474
//其实就是调用callback来处理这个属性,而args就是callback处理时要用到的参数
```

```
1475
1476                        //
当callback在处理某一个属性后返回false时,就不用对后面的属性进行处理了.
至于什么时候返回false, 这个由你的callback来决定,你喜欢.
1477                        if ( callback.apply( object[ name ], args ) ===
false )
1478                            break;
1479                }
1480
1481                //
else的话,说明object是类数组的对象(大部分情况下是jQuery对象),那就可以用
下标的方式来访问
1482                else
1483                    for ( ; i < length; )
1484                        if ( callback.apply( object[ i++ ], args ) ===
false )
1485                            break;
1486
1487            // A special, fast, case for the most common use of each
1488            }
1489
1490            // 如果没有 args 传进来
1491            else {
1492
1493                // length == undefined 说明 object不是类数组( array-like
)于是不能用下标的方法来访问
1494                if ( length == undefined ) {
1495                    for ( name in object )
1496                                            //call
函数第一个参数跟apply一样,起一种上下文的作用.
call后面的参数都作为callback的参数
1497
//可以看到,传参给callback,apply用了数组,而call则是一个一个传
1498                        if ( callback.call( object[ name ], name, object[
 name ] ) === false )
1499                            break;
1500                }
1501                //是类数组的对象,可以通过下标的方式来访问
1502                else
1503                    for ( var value = object[0];
1504                        i < length && callback.call( value, i, value )
!== false; value = object[++i] ){}
1505            }
1506
1507            return object;
1508        },
1509
1510        /*
1511         *
对属性值进行处理.取得正确的属性值.如,这个属性值是否要加上单位"px",
等等.
1512         *
1513         * elem - dom元素对象
1514         * value - 属性值
1515         * type - 如果有值就代表是样式属性名
1516         * i - dom元素在jQuery对象匹配元素集合中的索引
1517         * name - 属性名
1518         */
```

```
1519        prop: function( elem, value, type, i, name ) {
1520            // Handle executable functions
1521            // 如果属性值是function,
    就在elem上调用这个function(i作为参数),function处理的结果再作为value
1522            if ( jQuery.isFunction( value ) )
1523                value = value.call( elem, i );
1524
1525            // Handle passing in a number to a CSS
    property                    //exclude中表示了一些不需要加单位的属性值
1526            return value && value.constructor == Number && type ==
    "curCSS" && !exclude.test( name ) ?
1527                value + "px" :
1528                value;
1529        },
1530
1531        /**
1532         *
    jQuery.className命名空间,在这个命名空间上定义了一系列用来操作元素class
    Name属性的方法.
1533         *
    不过这个命名空间内的函数并不'对外开发',只是内部使用.对外开发的方法都是
    这些方法的包装方法.
1534         */
1535        className: {
1536            // internal only, use addClass("class")
1537            /**
1538             * 给一个普通的DOM元素添加一个类名
1539             * 内部使用,不对外公开
1540             *
1541             * @param {Object} elem
1542             * @param {Object} classNames
1543             */
1544            add: function( elem, classNames ) {
1545
1546                /* 方法体首先用split方法将classNames用空格'
    '分开,放到一个数组里.然后使用jQuery.each方法来遍历每一个
1547                 *
    className.如果元素没有这个className,那就把这个className接到元素的class
    Name的后面.否则,就什么事情
1548                 * 都不做.请看代码,其中包含了了一些没有说到的细节:
1549                 */
1550
1551                jQuery.each((classNames || "").split(/\s+/), function(i,
    className){
1552                    if ( elem.nodeType == 1/*NODE.ELEMENT_NODE*/ && !
    jQuery.className.has( elem.className, className ) )
1553                        elem.className += (elem.className ? " " : "") +
    className;
1554                });
1555            },
1556
1557            // internal only, use removeClass("class")
1558            /**
1559             * 去掉某个元素上的className
1560             * 内部使用,不对外公开
1561             * @param {Object} elem
1562             * @param {Object} classNames
1563             */
```

```
1564        remove: function( elem, classNames ) {
1565
1566            /*
    注意,以下代码中的jQuery.grep也是一个filter函数来的.可以参考jQuery.grep
    的中文注释. */
1567
1568            if (elem.nodeType == 1/* NODE.ELEMENT_NODE */)
1569                elem.className = classNames != undefined ?
1570                    jQuery.grep(elem.className.split(/\s+/), function
    (className){
1571
    //如果className在classNames的那个数组当中,就把它过滤掉,即不要.
1572                        return !jQuery.className.has( classNames,
    className );
1573                    }).
1574                    join(" ")//最后将剩下的结果用" "组合起来
1575                    :
1576                    "";
1577        },
1578
1579        // internal only, use hasClass("class")
1580        /**
1581         * 看看elem的类名中,有没有className指定的类名
1582         * 内部使用,不对外公开
1583         *
1584         * @param {HTMLElement} elem
1585         * @param {string} className
1586         */
1587        has: function( elem, className ) {
1588                                                //将elem的className用"
    "切分开来形成一个数组,然后用
1589
    //jQuery.inArray看看在不在里面.
1590            return jQuery.inArray( className, (elem.className || elem
    ).toString().split(/\s+/) ) > -1;
1591        }
1592    },
1593
1594    // A method for quickly swapping in/out CSS properties to get
    correct calculations
1595    /**
1596     * 这是一个很有意思的函数:
1597     *
    如果一个元素不处于一定的状态,那么直接获取它的某些属性值将会是不正确的.
1598     *
    如offsetWidth/offsetHeight在元素不可见的时候,直接获取它们的值是不正确
    的.
1599     * 因此先将元素的visibilty设置为 position: "absolute",
    visibility: "hidden", display:"block"
1600     * 然后计算所要样式值,最后将原来的样式属性设置回去.
1601     *
1602     * @param {HTMLElement} elem 普通的DOM元素
1603     * @param {Object} options
    对象.里面装着一些样式的键值对,用来设置计算样式值时的"环境"
1604     * @param {Function} callback
    这个函数在甚至设置环境后执行.当它执行完之后,元素的样式将会被重置.
1605     */
1606    swap: function( elem, options, callback ) {
```

```
1607            var old = {};
1608            // Remember the old values, and insert the new ones
1609            // 把旧的CSS样式保存下来,然后换上新的
1610            for ( var name in options ) {
1611                old[ name ] = elem.style[ name ];
1612                elem.style[ name ] = options[ name ];
1613            }
1614
1615
//换上了上新的样式值之后,赶紧'干活',获取想要的样式的值.注意这些值,在没有换上新的属性之前,是无法正确获取的.
1616            callback.call( elem );
1617
1618            // Revert the old values
1619            // 获取完毕,恢复原来的样式
1620            for ( var name in options )
1621                elem.style[ name ] = old[ name ];
1622        },
1623
1624        /**
1625         * 获取元素当前的CSS样式值.
1626         *
1627         * @param {HTMLElement} elem 元素
1628         * @param {string} name 样式属性名
1629         * @param {Object} force
一个布尔的开关,为true则直接获取元素的内联样式的值.获取失败就再尝试获取元素的计算样式.如果
1630         *
为false或者不传入这个值,则会在获取计算样式之前先看看内联样式上有没有name所指定的样式的值.
1631         */
1632        css: function( elem, name, force ) {
1633            //
如果是要获取width或者是height属性,需要特殊处理.特殊处理是基于如下的考虑:
1634            //
我们不是直接使用元素的width或者height属性来获取相应的值,而是使用了offsetWidth/Height. 这么获取widht/height
1635            // 就需要对结果进行'修剪'.因为会有border和padding的问题.
用具体看下面的代码.
1636            if ( name == "width" || name == "height" ) {
1637                var val,//这是最后要返回的结果
1638                    //
这个对象是传给待会要调用的jQuery.swap函数的,具体看下面代码或者参考jQuery.swap
1639                    props = { position: "absolute", visibility: "hidden", display:"block" },
1640
1641                    //
确定现在要获取的是width还是height.是width就要注意左右的padding和border;是height则要注意上下
1642                    which = name == "width" ? [ "Left", "Right" ] : [ "Top", "Bottom" ];
1643
1644                //
这个嵌套定义的内部函数用来计算元素的width/height.在使用offsetXX获取widht/height值的时候要注意将border
1645                // 和padding 减去.
```

```
1646            function getWH() {
1647                val = name == "width" ? elem.offsetWidth : elem.
    offsetHeight;
1648                var padding = 0, border = 0;
1649
    //which是一个数组,现在使用each遍历数组里面的每一个字符串.
1650                jQuery.each( which, function() {
1651                    //注意,现在this引用的是一个字符串,为 "Left",
    "Right" 中的一个, 或者"Top", "Bottom" 中的一个.
1652
1653                    /*
    下面计算padding和border(左右或者上下),计算的目的是想在最后获得的offset
    Width/offsetHeiht
1654                     * 中将他们减掉.有两点要说明:
1655                     * (1)
    使用offsetWidth而不用width是因为width是内联的,如果没有在HTML中设置widt
    h属性或者在JS代码
1656                     *
    中显式设置width的值,那么将无法获取元素的width/height属性值,即使在样式
    文件中设置了元素该样式的值.
1657                     *
    某些自适应的元素我们并没有显式地在任何地方设置它的width/height,但有时
    候又的确需要获取这些自适应
1658                     *
    元素的width/height值.而使用offsetXXX就可以满足这些场景下的应用需求.
1659                     * (2)
    offsetXXX的值是包含了padding和border的(注意,不包含margin),所以下面的代
    码需要把他们减掉
1660                     */
1661
1662                    padding += parseFloat(jQuery.curCSS( elem,
    "padding" + this, true)) || 0;
1663                    border += parseFloat(jQuery.curCSS( elem,
    "border" + this + "Width", true)) || 0;
1664                });
1665                val -= Math.round(padding + border);
1666            }
1667
    //如果元素有'visible'的样式,直接调用getWH获取所要的值.请参考jQuery.fn.
    is函数的注释.
1668            if ( jQuery(elem).is(":visible") )
1669                getWH();
1670
1671
    //如果没有visible,并不能确定能正确地获取到width或者height的值,为了防止
    出乱子,调用jQuery.swap函数先将
1672            //给元素设置上这样的属性{ position: "absolute",
    visibility: "hidden", display:"block" },再获取
1673
    //元素的width/height的值,那样就十拿九稳了.(注意,swap函数最后会重置元素
    的样式值).至于为什么用swap就能
1674            //避免乱子,可以参考jQuery.swap的中文注释.
1675            else
1676                jQuery.swap( elem, props, getWH );
1677
1678            //返回获取到的width/height值.
1679            return Math.max(0, val);
1680        }
```

```
1681
1682
1683
1684
        //如果不是获取width/height的值,就不用担心border/padding的问题,直接调用'幕后'的curCSS完成样式的获取.
1685            return jQuery.curCSS( elem, name, force );
1686        },
1687
1688        /**
1689         *
        获取元素当前正在使用的CSS属性值.这个方法是真正实现获取元素样式值的'幕后函数'.
1690         *
        它能够获取元素目前层叠和展现出来的样式的值,而不管这个值是内联的还是在别处(如嵌入式或css文件)层叠出来的.
1691         *
1692         * @param {HTMLElement} elem 要获取或设置这个元素的css
1693         * @param {string} name css属性的名字
1694         * @param {Object} force
        一个布尔的开关,为true则直接获取元素的计算样式的值.
1695         *
        如果为false或者不传入这个值,则会在获取计算样式之
1696         *
        前先看看内联样式上有没有name所指定的样式的值.
1697         */
1698        curCSS: function( elem, name, force ) {
1699            var ret, style = elem.style;
1700
1701            // A helper method for determining if an element's values are broken
1702            //
        翻译:一个用以判断元素的值是否"有损坏"(即是否正确)的辅助方法.
1703            /**
1704             * 该方法主要针对Safari.
1705             *
        在Safari中获取元素的color样式将会出现获取不到的情况.为了判断是否能够正确获取一个元素的计算样式
1706             * 编写了这个跨浏览器的辅助方法.
1707             *
1708             * @param {Object} elem 要判断的DOM元素.
1709             */
1710            function color( elem ) {
1711                /*
1712                 *
        COMP:在Safari中获取元素的color样式将会出现获取不到的情况.
1713                 *
        如果不是Safari就可以返回false了.因为他们不会存在这个问题.
1714                 */
1715                if ( !jQuery.browser.safari )
1716                    return false;
1717
1718                /*
1719                 * 如果代码能够运行到这里, 说明当前浏览器是Safari.
1720                 * 在Safari中获取元素的color样式将会出现获取不到的情况.
1721                 *
        使用下面这两行代码就是为了应付这个问题:(注意,下面的代码并不处理这个问题,只是起到一种报告的作用)
```

```
1722                 *
     如果getComputedStyle不能获取元素的计算样式,那么返回true(!ret),说明碰到了这个问题.因为连
1723                 *
     CSS2Properties都没有,那就更不用说在它上面获取color样式的值啦.
1724                 *
     如果getComputedStyle的确是返回了非空的CSS2Properties对象,那就看看能否获取非空白的color
1725                 * 属性.请看以下代码...
1726                 */
1727
1728                // defaultView is cached
1729                /* COMP:
1730                 *
     getComputedStyle函数的第二个属性在Mozilla和Firefox中实现对伪元素的查找.
1731                 *
     如可以将参数二设置为":after"或":before".这个参数在上述两种浏览器中不能省略.
1732                 *
     但是IE不支持这样的调用.一般情况下我们对伪元素也没有兴趣,于是为了省得出乱子,无论
1733                 * 在那一种浏览器中我都将第二个参数设为null.
1734                 */
1735                var ret = defaultView.getComputedStyle( elem, null );
1736                return !ret || ret.getPropertyValue("color") == "";
1737
1738            }
1739
1740            /* We need to handle opacity special in IE
1741             *
     COMP:IE中的透明度设置与获取跟w3c的不一样.这个地球人都知道了.
1742             */
1743            if ( name == "opacity" && jQuery.browser.msie ) {
1744                // 注意,
     参数style其实是elem.style属性.在这行的最后ret是属性'opacity'的值
1745                ret = jQuery.attr( style, "opacity" );
1746
1747                /*
1748                 *
     下面的代码就是通过ret的值是否为""来判断到底什么类型的浏览器从而返回相应的opacity值
1749                 *
     如果ret=="",说明style没有opacity这个属性,当前浏览器是IE(IE实现透明度使用filter),返回1,
1750                 * 100%的透明. 若ret!="",那就直接返回咯.
1751                 */
1752                return ret == "" ?
1753                    "1" :
1754                    ret;
1755            }
1756
1757
1758            /* Opera sometimes will give the wrong display answer, this
     fixes it, see #2037
1759             *
     COMP:我们似乎经常可以在jQuery的中都有一些很"神经"的代码,他们总是把style保存起来,然后设置一个
```

```
1760            *
新的值,又然后把保存起来的旧的值设置回去.这样子做是为了解决某些浏览器的在CSS渲染上的问题.像英文
1761            *
注释所说的那样,如果你有兴趣,可以到jQuery的官方网站上看看编号为#2037的这个issue，它地址是：
1762            * http://dev.jquery.com/ticket/2037
1763            * 也可以参照jQuery.swap函数的中文注释.
1764            */
1765           if ( jQuery.browser.opera && name == "display" ) {
1766               var save = style.outline;
1767               style.outline = "0 solid black";
1768               style.outline = save;
1769           }
1770
1771
1772
1773           /* Make sure we're using the right name for getting the float value
1774            *
COMP:其实这里也涉及到了一个兼容性问题.就是float这个样式属性在w3c中叫cssFloat,而在
1775            * IE中则叫styleFloat
1776            */
1777           if ( name.match( /float/i ) )
1778               name = styleFloat;
//styleFloat会根据浏览器的不同而选择使用'cssFloat(w3c)或者styleFloat(IE)'
1779
1780           // 如果没有为force指定值,那就不是"强迫直接获取计算样式值".
1781           // 那么就先返回style[]内的样式值.
注意,这里用style来获取的样式值只是元素内联样式而已
1782           //
不包括其他(样式文件等地方)设置的样式值.如果的确存在name所指定的内联样式的值,可以将ret返回了.
1783           if ( !force && style && style[ name ] )
1784               ret = style[ name ];
1785
1786           /* 如果没有那个样式的内联值,
说明这个样式属性应该到其他地方找. 下面的代码就是为了完这个目标 */
1787
1788           /* COMP:
1789            *
如果在上面那个if里面没有找到name所指定的style,那说明这个样式的值并不在内联样式中设置,那就要用到
1790            * 计算样式了,也就是下面两个else if
所要做的事情.defaultView.getComputedStyle是w3c的方法.
1791            * 而window.currentStyle则是IE的方法.
1792            */
1793           else if ( defaultView.getComputedStyle ) {//
<--检查是否存在defaultView.getComputedStyle
1794                                                   //
有就说明是在遵循w3c标准的浏览器里.
1795               // Only "float" is needed here
1796               if ( name.match( /float/i ) )
1797                   name = "float";
1798
1799               /*
```

```
1800                 * 把骆驼式的变量名改写成需要的css标准形式,如:
1801                 * backgroundColor -> background-color
1802                 * marginTop -> margin-top
1803                 */
1804                name = name.replace( /([A-Z])/g, "-$1" ).toLowerCase();
1805
1806
//获取elem的计算样式.这些样式或者定义在一个css文件中,又或者定义在其他的地方如<head>中的<style>标签内.
1807                var computedStyle = defaultView.getComputedStyle( elem, null );
1808
1809                /*
1810                 *
以上的computedStyle对象是一个CSS2Properties类的实例,通过这个对象的getPropertyValue
1811                 * 方法,传入一个样式的名字,
就能获取样式的值.下面这个if就用这个方法获取元素的计算样式.
1812                 *
注意if中的!color(elem),它的意思为"获取elem的color样式不会失败,getComputedStle能够正确报告自己的
1813                 * 计算样式值".
1814                 */
1815                if ( computedStyle && !color( elem ) )
1816
//经过if的测试,我们能够正确地获取元素的计算样式值.可以放心地获取name所指定的属性了.
1817                    ret = computedStyle.getPropertyValue( name );
1818
1819
1820
1821                // If the element isn't reporting its values properly in Safari
1822                // then some display: none elements are involved
1823                /*
1824                 *
运行这个else的大多数情况就是因为!color(elem)是false.这说明我们在获取元素的计算样式值会失败.
1825                 * 这里要说明一个重要的知识点:CSS(Cascade Style Sheet)是层叠样式表的意思,但是现在许多人都忽略了"层叠"
1826                 *
二字.实际上"层叠"是CSS的一个最基本的特征.一个元素当前展现出来的样式,其实是它自己及其祖先的样式一层一层
1827                 * 叠加的结果.
计算样式要获取的样式值就是这种叠加的结果.
所以当计算样式不正常的时候,解决方案自然而然地就是
1828                 * 从自己以及祖先上,一层一层地找问题.
1829                 *
于是下面的这个else就是从元素自己开始,遍历自身及其所有祖先.在遍历的过程中使用一种叫swap的方法,让元素的
1830                 *
display属性为block,这样就能正常获取某些样式的值(因为他们显示不正常多半是由于自己dispaly为none所造成).
1831                 * 而swap方法最后也会重置元素的display值.
1832                 */
1833                else {
1834                    var swap = [], stack = [], a = elem, i = 0;
1835
```

```
1836                // Locate all of the parent display: none elements
1837                //
    从自己开始,一直向上层找,发现不能正常获取其计算样式值的元素就把它放进stack中(压栈)
1838                for ( ; a && color(a); a = a.parentNode )
1839                    stack.unshift(a);//在数组头"压入"元素
1840
1841                // Go through and make them visible, but in reverse
1842                // (It would be better if we knew the exact display type that they had)
1843                /*
1844                 *
    在以下的for循环里,我们从最顶层的祖先开始,倒着顺序(即从最外层到最内层)设置元素的display让其为可见(block),
1845                 * 并且记录下每一个元素原先的display类型,待会设置回去.
1846                 * 这种设置display为block,然后do something,最后重置display的方法,John Resig(jQeury作者,我师父)
1847                 * 称之为 Swap.
    在jQuery源代码中,也存在这样一个叫jQuery.swap的内部方法.
1848                 */
1849                for ( ; i < stack.length; i++ )
1850                    /*
    在上一个for中,我们首先将元素自己加进了stack中,而不管它是否经过color了的测试.因此需要在这里用
1851                     * color函数判断是不是每个元素都有问题.
    注意,变量a是有可能在其上正确地获取计算样式的.所以
1852                     * swap.length == stack.length 或者 swap.length == stack.length - 1.
1853                     * 所以造成了后面的代码出现swap[ stack.length - 1 ]是否为null的判断.如果是null,说明元素
1854                     *
    的计算样式正常(因为不正常才会在swap中有"案底"),可以放心获取.
1855                     */
1856                    if ( color( stack[ i ] ) ) {
1857                        swap[ i ] = stack[ i ].style.display;
1858                        stack[ i ].style.display = "block";
1859                    }
1860
1861                // Since we flip the display style, we have to handle that
1862                // one special, otherwise get the value
1863                /*
1864                 *
    我们采用了让display为block的方法来获取正常的计算样式,但如果要获取的计算样式正是display自己呢?
1865                 *
    以下代码是说,如果要获取的计算样式名不是display或者是display但元素的计算样式没有broken,那么可以
1866                 *
    用computedStyle放心获取.如果是display并且在swap中有"案底"(!=null),就让最后的返回结果ret为
1867                 * "none". 至于为什么是none,这个我保留意见(<-TODO).
1868                 */
1869                ret = name == "display" && swap[ stack.length - 1 ] != null ?
1870                    "none" :
1871                    ( computedStyle && computedStyle.getPropertyValue( name ) ) || "";
```

```
1872
1873                // Finally, revert the display styles back
1874                // 翻译:最后,恢复原来的样式
1875                for ( i = 0; i < swap.length; i++ )
1876                    if ( swap[ i ] != null )
1877                        stack[ i ].style.display = swap[ i ];
1878            }//end else
1879
1880            // We should always get a number back from opacity
1881            /*
经过上面的代码所要的计算样式都应该有结果了.这些结果要么是正常的值要么就是""
1882             *
特别地对opacity进行处理一下,如果获取它的结果是"",那至少让它是"1",不透明.
1883             */
1884            if ( name == "opacity" && ret == "" )
1885                ret = "1";
1886
1887        }
1888        /*
1889         * 如果代码运行这个else
if,那说明当前浏览器是IE,那就使用IE特有的方法来获取计算样式.
1890         */
1891        else if ( elem.currentStyle ) {
1892
//camelCase是说类似"paddingTop"第二个单词首字母大写的变量名书写形式
1893            //以下代码是将属性名称改写成camelCase的形式
1894            var camelCase = name.replace(/\-(\w)/g, function(all,
letter){
1895                return letter.toUpperCase();
1896            });
1897            // 使用传入的属性名称试一试,
如果不行,就使用camelCase的变量名形式再试一试.
1898            ret = elem.currentStyle[ name ] || elem.currentStyle[
camelCase ];
1899
1900            // From the awesome hack by Dean Edwards
1901            //
http://erik.eae.net/archives/2007/07/27/18.54.15/#comment-102291
1902
1903            /*
1904             * TODO:以下if中的代码比较匪夷所思,他的作者Dean
Edwards(Resig也是'抄'来的)并没有说明其原理.
1905             * 我分析了好久都不明玄妙,希望高人指点!
1906             */
1907
1908            // If we're not dealing with a regular pixel number
1909            // but a number that has a weird ending, we need to
convert it to pixels
1910            /* 直接翻译:
如果获取到的结果不是一个正常的pixel值(如,100,没有px结尾)而是一个怪异结尾的数字,我需要将它
1911             * 转换为正常pixels.
1912             */
1913            if ( !/^\d+(px)?$/i.test( ret ) && /^\d/.test( ret ) ) {
//正则表达式就不解释了,写起来就罗嗦了.
1914                // Remember the original values
```

```
1915                    //
    暂时把style.left,runtimeStyle.lfet样式值保存起来,等获取到需要的值的时
    候再设回去.
1916                    var left = style.left, rsLeft = elem.runtimeStyle.
    left;
1917
1918
1919                    /*
    在IE中,元素的runtimeStyle与style的作用相同,但优先级更高.
1920                     *
    在w3c标准中document.defaultView.getOverrideStyle是与之相对应的方法,
1921                     *
    但是由于Gecko核心的浏览器并没有实现这个标准,所以这意味着这种方法仅对IE
    有效.
1922                     */
1923
1924                    // Put in the new values to get a computed value out
1925                    elem.runtimeStyle.left = elem.currentStyle.left;
1926                    style.left = ret || 0;
    //style在函数的最开始有定义:style = elem.style
1927                    ret = style.pixelLeft + "px";
1928
1929                    // Revert the changed values
1930                    // 恢复原来的样式
1931                    style.left = left;
1932                    elem.runtimeStyle.left = rsLeft;
1933                }
1934            }
1935
1936            return ret;
1937        },
1938
1939
1940        /**
1941         * 如果elems是Number类型,则变成数字字符;
1942         * 如果elems是XHTML字符串,则将这些字符变成真正的DOM Element
    然后把这些元素存储在匹配元素集合里面.
1943         *
    在jQuery对象的构造函数中,当传入的字符串是HTML字符串时,jQuery将会调用本
    函数来将这些字符串转化成为DOM Element.
1944         *
1945         * @param {string} elems - 它是一个字符串.
1946         * @param {HTML Element} context -
    elem所处的上下文.在本函数中,这个context必须为一个与elems有包含关系的do
    cument.
1947         */
1948        clean: function( elems, context ) {
1949            var ret = [];
1950            context = context || document;
1951            // COMP: !context.createElement fails in IE with an error
    but returns typeof 'object'
1952            //
    作者本来想使用'if(!context.createElement)'来代替下面这个'if',
    但是因为在IE中这个表达式不能达到目的, 于是使用了现在这个
1953            // 表达式.
1954            if (typeof context.createElement == 'undefined')
    //如果context没有context.createElement方法, 那必须让context
1955
```

```
        //至少是一个拥有createElement的元素. 之所以不直接让context等于
1956
        //document是因为考虑到jQuery的对象不仅仅是HTML, 还有XML.
1957                context = context.ownerDocument || context[0] && context[
        0].ownerDocument || document;
1958
1959            /* 对elems里的每一个元素都调用一个匿名函数进行处理
1960             * 这个函数进行的处理是:
1961             * 如果这个元素是Number类型,则把它变成数字字符.
1962             * 如果这个元素是XHTML string,
        那把string中的">"和"<"之间的文本去掉,然后将它变dom对象并获取其内的子节
        点数组(childNodes),最后把这个数组装进一个
1963             * 结果集里面
1964             *
1965             *
1966             * 下面就是上述功能的实现. 里面包括了一些上面没有讲到的细节.
1967             */
1968            jQuery.each(elems, function(i, elem){
1969                if ( !elem )//如果elem是空的,那就不用干活了.
1970                    return;
1971
1972                if ( elem.constructor == Number )
        //如果是数字,则通过与''进行'+'运算将数字变成数字字符.
1973                    elem += '';
1974
1975                // Convert html string into DOM nodes
1976                // 翻译:将HTML 字符串转化成DOM节点
1977                if ( typeof elem == "string" ) {
1978                    // Fix "XHTML"-style tags in all browsers
1979                                                            //
        参数说明: all - 匹配到的整个字符串 ; front -
        match的第一个分组'<标签名';
1980
        //          tag - 第二个分组(在这里就是tag的名称)
1981                    elem = elem.replace(/(<(\w+)[^>]*?)\/>/g, function(
        all, front, tag){
1982                        return tag.match(
        /^(abbr|br|col|img|input|link|meta|param|hr|area|embed)$/i) ?
1983                            all ://
        如果是上面列出的tag(他们都是单标签),直接把匹配到的标签返回
1984                            front + "></" + tag + ">";
        //如果不是上面所列的单标签,就把他们变成空的双标签并返回
1985                    });
1986
1987                    // Trim whitespace, otherwise indexOf won't work as
        expected
1988                    var tags = jQuery.trim( elem ).toLowerCase(),
1989                        div = context.createElement("div");
1990                        /* 现在对以上的这个div进行说明:
1991                         *
        这个div只是一个临时的容器而已.目的是等下将合适的html
        string装进这个div里面(通过innerHTML方法),
1992                         * 好让string变成DOM元素.这个是将HTML
        string转化成为DOM元素比较流行和唯一的做法.
1993                         */
1994
1995
1996                    // 对下面wrap的说明:
```

```
1997                    //
要elem的标签加上一个"外壳",而wrap就是用来"包住"elem的html标签
1998                    // 下面将检查tags里面是否含有所列举的标签.
如果有就把wrap赋值为对应的"外壳"标签,等下使用.
1999                    // 对下面你看到的数组进行说明:
2000                    // 数组下标为0的元素表示这个"外壳"含有几层,
元素1、2表示"外壳"的开始和结束标签
2001                    //
请注意运算符'&&'在JavaScript中的'妙用'(参见'attr'函数的中文注释.提示,c
trl + 'F',搜索'attr:')
2002                    var wrap =
2003                        // option or optgroup
2004                        !tags.indexOf("<opt") &&
2005                        [ 1, "<select multiple='multiple'>", "</select>"
] ||
2006
2007                        !tags.indexOf("<leg") &&
2008                        [ 1, "<fieldset>", "</fieldset>" ] ||
2009
2010                        tags.match(/^<(thead|tbody|tfoot|colg|cap)/) &&
2011                        [ 1, "<table>", "</table>" ] ||
2012
2013                        !tags.indexOf("<tr") &&
2014                        [ 2, "<table><tbody>", "</tbody></table>" ] ||
2015
2016                        // <thead> matched above
2017                        (!tags.indexOf("<td") || !tags.indexOf("<th")) &&
2018                        [ 3, "<table><tbody><tr>",
"</tr></tbody></table>" ] ||
2019
2020                        !tags.indexOf("<col") &&
2021                        [ 2, "<table><tbody></tbody><colgroup>",
"</colgroup></table>" ] ||
2022
2023                        // IE can't serialize <link> and <script> tags
normally
2024                        jQuery.browser.msie &&
2025                        [ 1, "div<div>", "</div>" ] ||
2026
2027                        [ 0, "", "" ];
2028
2029                    // Go to html and back, then peel off extra wrappers
2030                    //
用这个"外壳"把elem包起来.注意,通过innerHTML的运算之后,HTML
string已经成为了实实在在的DOM元素.
2031                    div.innerHTML = wrap[1] + elem + wrap[2];
2032
2033                    // Move to the right depth
2034                    //
通过循环,让div指向被包裹的elem元素的上一层.举个简单一点的例子,如果我们
传入的HTML string是 '<option>linhuihua.com</option>'
2035                    //
那么经过innerHTML之后,div的DOM层次是(注意div再也不是字符串,人家现在是
一个堂堂正正的DOM元素):
2036                    /* <select>
2037                     *      <option>linhuihua.com</option>
2038                     * </select>
2039                     * 进过下面这个while循环之后,
```

```
'div'这个引用将会指向'select'这一层
2040                     */
2041                    while ( wrap[0]-- )
2042                        div = div.lastChild;
2043
2044
2045                    /*
COMP:如果世界上没有IE浏览器,那么下面这个'if'就可以省略了...
2046                     * 看以下代码的时候,可以首先省略下面的这个'if':
2047                     * 经过上面的while处理之后, 现在要想获得HTML
string所对应的DOM元素只要用div.childNodes里面的元素组成一个
2048                     *
array然后返回即可(childNodes只是类数组对象,它没有数组的排序等算法).
很可惜的是,IE会给<table>元素内自动
2049                     *
加上<tbody>,这样就破坏了我们企图通过childNodes获得我们想要元素的美梦.
下面就写一个if语句来处理IE的这个问题.
2050                     */
2051                    // Remove IE's autoinserted <tbody> from table
fragments
2052                    // IE会在table 内自己加上<tbody>, 现在要把它去掉
2053                    if ( jQuery.browser.msie ) {
2054
2055                        // String was a <table>, *may* have spurious
<tbody>
2056                        var tbody = !tags.indexOf("<table") && tags.
indexOf("<tbody") < 0 ?
2057                                div.firstChild && div.firstChild.childNodes :
//如果没有table标签并且也没有tbody标签,就返回div下的所有子节点,用tbody
装着
2058
2059                            // String was a bare <thead> or <tfoot>
2060                            wrap[1] == "<table>" && tags.indexOf("<tbody"
) < 0 ?//如果是table标签,但其内没有tbody标签
2061                                    div.childNodes :
//(true)就返回标签下的所有子元素
2062                                    [];//(false)就返回一个空的集合
2063
2064                        for ( var j = tbody.length - 1; j >= 0 ; --j )
2065                                    // 如果tbody[j]是一个tbody标签元素
&& 这个tbody元素有子元素
2066                            if ( jQuery.nodeName( tbody[ j ], "tbody" )
&& !tbody[ j ].childNodes.length )
2067                                    tbody[ j ].parentNode.removeChild( tbody[
 j ] );//叫自己的parent把自己删除掉
2068
2069                        // IE completely kills leading whitespace when
innerHTML is used
2070                        // COMP:在使用innerHTML向元素注入HTML时,
如果这些HTML由一个whitespace打头(开始),IE会把这个whitespace给"kill"掉
2071                        // 检查是不是属于这种情况,
如果是就把这个whitespace插回去
2072                        if ( /^\s/.test( elem ) )
2073                            div.insertBefore( context.createTextNode(
elem.match(/^\s*/)[0] ), div.firstChild );
2074
2075                    }
2076
```

```
2077
2078                // 处理完IE会自己加tbody的bug之后,就可以把结果返回了
2079                // 由于childNodes并不是一个真正的数组,
因此需要调用jQuery.makeArray来将它转换成为一个数组.详细参见jQuery.make
Array
2080                // 的中文注释.
2081                elem = jQuery.makeArray( div.childNodes );
2082            }//end if( type of elem == "string")
2083
2084
2085            //
在最后返回之前如果发现结果集里面没有元素,并且elem又不是(form或者select
)
2086            if ( elem.length === 0 && (!jQuery.nodeName( elem, "form"
 ) && !jQuery.nodeName( elem, "select" )) )
2087                return;//什么也不做了,返回.
不用把这个元素加到结果集里面
2088
2089            // 下面两种情况都要把elem加到结果集里面
2090            if ( elem[0] == undefined || jQuery.nodeName( elem,
"form" ) || elem.options )
2091                ret.push( elem );
2092            else
2093                ret = jQuery.merge( ret, elem );
2094
2095        });// end jQuery.each();
2096
2097        return ret;
2098    },
2099
2100    /**
2101     * 获取或设置属性值.没有传入value则被视为获取属性值的操作
2102     *
2103     * @param {HTMLElement} elem
2104     * @param {string} name
2105     * @param {string} value
2106     */
2107    attr: function( elem, name, value ) {
2108        // don't set attributes on text and comment nodes
2109        if (!elem || elem.nodeType == 3 || elem.nodeType == 8)
2110            return undefined;
2111
2112        var notxml = !jQuery.isXMLDoc( elem ),
2113            // Whether we are setting (or getting)
2114            set = value !== undefined,//我们是seting呢还是getting
2115            msie = jQuery.browser.msie;//是否是IE
2116
2117        // Try to normalize/fix the name
2118        // 一些属性名字在Js中的表述并不是原来的属性名字.
如class,在JavaScript中就是className. 所以要对这种情况进行处理
2119        name = notxml && jQuery.props[ name ] || name;
2120
2121        // Only do all the following if this is a node (faster for
style)
2122        // IE elem.getAttribute passes even for style
2123        // 我们只对有tagName的元素进行属性的访问或者设置
2124        if ( elem.tagName ) {
2125
```

```
2126            // These attributes require special treatment
2127            var special = /href|src|style/.test( name );
2128
2129            // Safari mis-reports the default selected property of a
    hidden option
2130            // Accessing the parent's selectedIndex property fixes it
2131            // COMP:这是Safari的一个bug
2132            if ( name == "selected" && jQuery.browser.safari )
2133                elem.parentNode.selectedIndex;
2134
2135            // If applicable, access the attribute via the DOM 0 way
2136            // 如果elem的属性中有name所指示的属性 &&
    elem不是XML类型节点 && 不是要特殊对待的属性(href/src/style)
2137            if ( name in elem && notxml && !special ) {
2138                if ( set ){
2139                    // We can't allow the type property to be
    changed (since it causes problems in IE)
2140                    //
    在IE中不能改变input元素的type属性，不然就会出乱子。所以只要是修改input
    的type属性的操作都给它一个Exception!
2141                    if ( name == "type" && jQuery.nodeName( elem,
    "input" ) && elem.parentNode )
2142                        throw "type property can't be changed";
2143
2144                    // 好了, 可以设置属性了
2145                    elem[ name ] = value;
2146                }
2147
2148                // browsers index elements by id/name on forms, give
    priority to attributes.
2149                // 设置完之后还要把设置成功后的值返回
2150
2151                // 如果是elem是一个form && elem有这个属性
2152                if( jQuery.nodeName( elem, "form" ) && elem.
    getAttributeNode(name) )
2153                    return elem.getAttributeNode( name ).nodeValue;
    //返回这个属性的值
2154
2155                // 程序运行到这里, 说明元素不是一个form元素,
    那直接就把元素的值返回
2156                return elem[ name ];
2157            }
2158
2159            // 程序能运行到这里, 说明 elem 并没有 name
    所指示的属性或者name是一个特殊属性(href/src/style)
2160            // 如果是IE浏览器 && 不是 XML节点 && name == "style"
2161            if ( msie && notxml &&  name == "style" )
2162                return jQuery.attr( elem.style, "cssText", value );
2163
2164            if ( set )
2165                // convert the value to a string (all browsers do
    this but IE) see #1070
2166                // 如果属性是一个非string的值,
    除IE外所有的浏览器都能很好地工作.
    所以为了让大家都能很好地工作,使用""+vlaue将value变成一个string
2167                elem.setAttribute( name, "" + value );
2168
2169            /* set完值之后就要把刚刚设置的值返回 */
```

```
2170
2171            // 获取IE的href/src/style属性需要进行特别步骤
2172            var attr = msie && notxml && special
2173                    // Some attributes require a special call on IE
2174                    ? elem.getAttribute( name, 2 )//
    getAttribute在网上说只有一个参数. 而IE可以用两个参数.果然是"special
    call on IE"
2175                    : elem.getAttribute( name );
2176
2177
2178
2179            // Non-existent attributes return null, we normalize to
    undefined
2180            // 如果返回 undefined就说明属性设置失败了
2181            return attr === null ? undefined : attr;
2182        }
2183
2184        // elem is actually elem.style ... set the style
2185        // 前面有一行代码: " jQuery.attr( elem.style, "cssText",
    value ); " ,以下的代码就是处理这种情况的了
2186
2187        // IE uses filters for opacity
2188        // 如果是IE的opacity滤镜
2189        if ( msie && name == "opacity" ) {
2190            if ( set ) {
2191                // IE has trouble with opacity if it does not have
    layout
2192                // Force it by setting the zoom level
2193                /*
2194                 * 原文翻译:
    如果元素没有layout,那么IE在处理opacity的时候就会出现问题.设置zoom值强
    迫它拥有layout.
2195                 *
2196                 * 一定有人不明白什么是layout的:
2197                 * layout
    是IE独有的概念.(请相信我,这个layout是不是你脑海里面的那个layout.总之,
    你可以把它当作新名词来理解就可以了):
2198                 * 简单地说, 在IE中,一个元素如果拥有layout,
    那么它就可以控制自己及其子元素的尺寸和位置.注意,在其他的浏览器中并没有
    一个元素拥有layout
2199                 * 这样的概念. layout的存在是当前IE不少Bug的根源.
    关于layout的更多内容,请参阅《精通CSS——高级Web标准解决方案》(CSS
    Mastery Advanced
2200                 * Standards Solutions)一书.
2201                 *
2202                 *
    下面的zoom属性是Mirosoft独有的属性,通过设置它, 能迫使一个元素具有layou
    t.
2203                 */
2204                elem.zoom = 1;
2205
2206                // Set the alpha filter to set the opacity
2207                elem.filter = (elem.filter || "").replace(
    /alpha\([^)]*\)/, "" ) +
2208                    (parseInt( value ) + '' == "NaN" ? "" :
    "alpha(opacity=" + value * 100 + ")");
2209            }
2210
```

```
2211                /* set完之后,就把刚刚set的值返回.
    直接获取属性值,也是执行下面的代码 */
2212
2213                return elem.filter && elem.filter.indexOf("opacity=") >=
    0 ?
2214                    (parseFloat( elem.filter.match(/opacity=([^)]*)/)[1]
    ) / 100) + '':
2215                    "";
2216            }
2217
2218            // 程序能运行到这里,说明并非要设置或者获取IE的opacity属性.
2219            // 就是要设置或获取普通的Style属性.
2220                                        //正则表达式后边的ig的含义: i -
    忽略大小写区别  ; g - 匹配所有可能的子串,一个也不要放过
2221                                        // 匿名函数中函数参数说明:
    参数all - 匹配到的整个字符串 ; 参数letter -
    匹配到的字符串的字母部分(第1个分组)
2222            name = name.replace(/-([a-z])/ig, function(all, letter){
2223                return letter.toUpperCase();
    //把匹配到的字符变成大写的,实际上是想做这样的效果:"margin-Top" ->
    "marginTop"
2224            });
2225
2226            if ( set )
2227                elem[ name ] = value;
2228
2229            return elem[ name ];
2230        },
2231        /**
2232         * 普通的trim函数,和Java String的trim函数的作用一样:
2233         * 将字符串头尾的" "空字符去掉
2234         *
2235         * @param {string} text 需要整理的字符串
2236         */
2237        trim: function( text ) {
2238            return (text || "").replace( /^\s+|\s+$/g, "" );
2239        },
2240
2241        /**
2242         * 用一个Array克隆另外一个内容一致的数组
2243         *
    注意这种克隆并非真正意义上的克隆.因为如果源数组内的元素是引用的话,在源
    数组上对引用内容的任何修改都会反映到目标数组上.
2244         * @param {Array} array 源数组.
2245         */
2246        makeArray: function( array ) {
2247            var ret = [];
2248
2249            if( array != null ){
2250                var i = array.length;
2251                // the window, strings and functions also have 'length'
2252                //
    jQuery作者Resig(也就是我师父..)告诉了我们一个天大的秘密:window,string,
    function都有'length'属性哦.
2253                // 如果是这些元素,那么就把这些元素作为数组的第一个元素.
2254                if( i == null || array.split || array.setInterval ||
    array.call )
2255                    ret[0] = array;
```

```
2256            else
2257                while( i )
//这里ret[--i]会比array[i]先运行哦,不然就会出现数据越界的问题.
2258                    ret[--i] = array[i];
2259        }
2260        //最后返回这个新的数组.
2261        return ret;
2262    },
2263
2264    /**
2265     * 检测一个元素是否在提供的array之内.返回元素在数组中的下标.
2266     *
2267     * @param {HTMLElement} elem
2268     * @param {Array} array
2269     */
2270    inArray: function( elem, array ) {
2271        for ( var i = 0, length = array.length; i < length; i++ )
2272        // Use === because on IE, window == document
2273        // COMP:在IE中表达式(window == document)返回true...所以使用===
2274            if ( array[ i ] === elem )
2275                return i;
2276
2277        return -1;
2278    },
2279
2280    /*
2281     * 把两个数组拼接起来(将第二个数组接到第一个数组的尾巴上)
2282     */
2283    merge: function( first, second ) {
2284        // We have to loop this way because IE & Opera overwrite the
length
2285        // expando of getElementsByTagName
2286        var i = 0, elem, pos = first.length;
2287
2288        // Also, we need to make sure that the correct elements are
being returned
2289        // (IE returns comment nodes in a '*' query)
2290        // 翻译:在一个使用'*'的选择器中,IE会返回注释节点.
2291        if ( jQuery.browser.msie ) {
2292            while ( elem = second[ i++ ] )
2293                if ( elem.nodeType != 8 )//NodeType == 8 是comment节点
2294                    first[ pos++ ] = elem;
2295
2296        } else
2297            while ( elem = second[ i++ ] )
2298                first[ pos++ ] = elem;
2299
2300        return first;
2301    },
2302
2303    /**
2304     * 返回一个数组中不重复的所有元素(用一个新的数组装着返回)
2305     *
2306     * @param {Object} array
2307     */
2308    unique: function( array ) {
2309        var ret = [], //结果集
2310            done = {};//用这个对象记录某个元素是否被处理过
```

```
2311
2312            try {
2313
2314                for ( var i = 0, length = array.length; i < length; i++ )
     {
2315
    //使用jQuery.data函数获取元素的id.即使元素没有id,jQuery函数也会为元素
    新建一个id.
2316                    var id = jQuery.data( array[ i ] );
2317
2318
    //如果这个id是第一出现,就在done里面记录下来,表示已经处理过.然后把元素
    放进结果集中等待返回.
2319                    if ( !done[ id ] ) {
2320                        done[ id ] = true;
2321                        ret.push( array[ i ] );
2322                    }
2323                }
2324
2325            } catch( e ) {
2326                //发生任何异常就直接把原数组返回.
2327                ret = array;
2328            }
2329
2330            return ret;
2331        },
2332
2333        /**
2334         * 使用过滤函数过滤数组元素。
2335         * 此函数至少传递两个参数:
2336         * elems - 待过滤数组
2337         * callback - 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。
2338         * inv -
    第三个参数是invert的意思,即过滤的逻辑跟第二个参数指定的函数的逻辑相反.
2339         *
    如,第二个参数的逻辑如果为选择大于0的元素,那么如果三个参数为true,则将小
    于0的元素选择出来
2340         *
2341         * @param {Object} elems
2342         * @param {Object} callback
2343         * @param {Object} inv
2344         */
2345        grep: function( elems, callback, inv ) {
2346            var ret = [];
2347
2348            // Go through the array, only saving the items
2349            // that pass the validator function
2350            for ( var i = 0, length = elems.length; i < length; i++ )
2351                if ( !inv != !callback( elems[ i ], i ) )
2352                    ret.push( elems[ i ] );
2353
2354            return ret;
2355        },
2356
2357        /**
2358         * 用第二个参数 callback
    提供的映射函数处理第一个参数中的每一个元素,
```

```
    将处理结果用一个新的数组装起来并返回
2359         *
2360         * @param {Array} elems 类数组或者数组
2361         * @param {Function} callback 映射处理函数
2362         */
2363        map: function( elems, callback ) {
2364            var ret = [];
2365
2366            // Go through the array, translating each of the items to
    their
2367            // new value (or values).
2368            // 翻译: 遍历数组(也可以是类数组对象),
    把其中的每一个元素转换为他新值.
2369            for ( var i = 0, length = elems.length; i < length; i++ ) {
2370                var value = callback( elems[ i ], i );
2371
2372                // value不是 null，说明处理成功，把它加入新的数组里面去
2373                if ( value != null )
2374                    ret[ ret.length ] = value;
2375            }
2376
2377            //
    返回一个新的数组，这个数组中的每个元素都是由原来数组中的元素经callback
    处理后得来
2378            return ret.concat.apply( [], ret );
2379        }
2380    });
2381
2382
2383    //我们将会使用navigator.userAgent来判断当前用户代理是哪一款浏览器
2384    var userAgent = navigator.userAgent.toLowerCase();
2385
2386    //Figure out what browser is being used
2387    //COMP:userAgent并不一定是可靠的.
2388    //在下面的代码里就使用userAgent来判断当前浏览器了.
2389    //可以在下面看到一些挺有趣的结果:比如写着是IE但可能是opera浏览器,写着M
    ozilla但也有有可能是IE浏览器...
2390    jQuery.browser = {
2391        version: (userAgent.match( /.+(?:rv|it|ra|ie)[\/: ]([\d.]+)/ ) ||
     [])[1],
2392        safari: /webkit/.test( userAgent ),
2393        opera: /opera/.test( userAgent ),
2394        msie: /msie/.test( userAgent ) && !/opera/.test( userAgent ),
2395        mozilla: /mozilla/.test( userAgent ) && !/(compatible|webkit)/.
    test( userAgent )
2396    };
2397    //float这个样式属性在IE和在其他现代标准浏览器中的名字有些不一样.
2398    var styleFloat = jQuery.browser.msie ?
2399        "styleFloat" ://如果是IE浏览器
2400        "cssFloat";//如果是其他的浏览器
2401
2402    //通过extend函数让jQuery的名字空间具有以下一些属性:
2403    //(1) boxModel:当前浏览器是否使用标准的盒子模型.
2404    //(2) props
    :一些特殊的样式名称在JavaScript被改成了其他的叫法,目的是不与一些JS现有
    的保留字冲突,如for,这是流程控制
2405    //语言的关键字,那么在JavaScript中就是用了htmlFor来代替它.
2406    jQuery.extend({
```

```
2407        // Check to see if the W3C box model is being used
2408        //
不是ie就一定使用了w3c的盒子模型.如果是ie,那就要看看有没有使用CSS1Compat,如果有的话,也是w3c的盒子模型.
2409        boxModel: !jQuery.browser.msie || document.compatMode == "CSS1Compat",
2410
2411        props: {
2412            "for": "htmlFor",
2413            "class": "className",
2414            "float": styleFloat,
2415            cssFloat: styleFloat,
2416            styleFloat: styleFloat,
2417            readonly: "readOnly",
2418            maxlength: "maxLength",
2419            cellspacing: "cellSpacing"
2420        }
2421    });
2422
2423    /*
2424     *
这里调用jQuery.each函数,逐个访问下面罗列出来的各个函数.each函数的第二个参数(它是一个函数)对他们进行一定的包装
2425     *
之后,将他们'接'到每一个jQuery对象上.让每一个jQuery对象都具有这些包装方法.
2426     * 可以看到,这组函数与DOM操作有关.具体细节请接下来慢慢欣赏:
2427     */
2428    jQuery.each({
2429        parent: function(elem){return elem.parentNode;},
2430        parents: function(elem){return jQuery.dir(elem,"parentNode");},
2431        next: function(elem){return jQuery.nth(elem,2,"nextSibling");},
2432        prev: function(elem){return jQuery.nth(elem,2,"previousSibling");},
2433        nextAll: function(elem){return jQuery.dir(elem,"nextSibling");},
2434        prevAll: function(elem){return jQuery.dir(elem,"previousSibling");},
2435        siblings: function(elem){return jQuery.sibling(elem.parentNode.firstChild,elem);},
2436        children: function(elem){return jQuery.sibling(elem.firstChild);},
2437        contents: function(elem){return jQuery.nodeName(elem,"iframe")?elem.contentDocument||elem.contentWindow.document:jQuery.makeArray(elem.childNodes);}
2438    },
2439    /*
each函数将遍历上面列出的所有方法.然后对上面的每一个方法调用以下函数进行处理.处理的内容是:
2440     *
将每一个方法包装一下,然后将这个包装过后的方法'接'到每一个jQuery对象上.让每一个jQuery对象都具有这些包装方法.
2441     */
2442    function(name, fn){
2443
2444
//在这个scope中的代码里,this指向每一个上面列出的函数,如'parent'.而在下面的代码中,function(selector)内的
2445        //的this指的是一个jQuery对象.
2446
```

```
2447        /* 以下将进行包装fn的工作.实际上,包装的内容为:
2448         * (1)
使用fn对jQuery对象内匹配元素集合中的每一个元素进行处理.处理后的结果用一个数组返回.
2449         * (2) 使用selector对象(1)所得的数组进行过滤
2450         * (3) 将(2)处理后的数组替换掉jQuery对象原来匹配元素集合.
2451         * 请看以下具体代码,里面有一些这里没有提到的细节.
2452         */
2453
2454        /*
以下这个函数就是所谓的包装函数了.这个包装函数最后将成为jQuery对象的一个方法.如我们要包装parent函数:
2455         *
parent函数返回元素的parent.而这个函数经过包装之后,功能就变成了返回一个集合,而这个集合就是jQuery对象
2456         *
内匹配元素集合中每一个元素的parent所组成的数组.而在返回这个数组之前,还需要经过selector的过滤.
2457         */
2458        jQuery.fn[ name ] = function( selector ) {
2459
//调用map函数,让jQuery对象内匹配元素集合中每一个元素都进过fn的处理,处理后的结果集中起来,放到一个数
2460            //组中返回给ret.
也许jQuery.map的中文注释能够帮助你理解这一步...
2461            var ret = jQuery.map( this, fn );
2462
2463            //经过fn的处理之后,如果有selector就是使用selector进行过滤.
2464            if ( selector && typeof selector == "string" )
2465                /*
调用过jQuery模块的核心函数multiFilter对ret进行过滤.注意,multiFilter接收三个参数.而这里只给
2466                 *
它传入了两个参数.这样调用的结果跟传入三个参数且第三个参数为false是一样的,即selector所描述的元素
2467                 *
全部都要留下来作为结果.具体可以参考jQuery.multiFilter的中文注释.
2468                 */
2469                ret = jQuery.multiFilter( selector, ret );
2470
2471            /* 返回结果之前,还要进行处理:
2472             * (1)
使用unique函数将ret中重复的元素去掉.(参考jQuery.unique函数)
2473             * (2)
由于最终目的是想要用结果集替换jQuery对象内的匹配元素集合,也即本函数将会修改匹配元素集合,故需要在这里
2474             *
设置一个'恢复点',以便能在需要的时候调用jQuery.fn.end函数来恢复.(同时请参考jQuery.fn.pushStack)
2475             */
2476            return this.pushStack( jQuery.unique( ret ) );
2477        };
2478    });
2479
2480    /* 以下的each调用跟上面那个each调用的思路一致.
2481     *
不过现在有些小改变.这次利用这个each调用在原有函数的基础之上稍作包装,让这些包装成为原有函数的'逆调用'.
```

```
2482     *
         如原来的a.apppend(b)是把b追加到a的childNodes里,而现在则利用append,来定义一个appendTo函数,它
2483     *
         在作用是若a.appendTo(b),那么就是把a追加到b的childNodes里.其他函数类推.
2484     *
2485     * 可以看到这组函数与DOM操作有关.
2486     */
2487    jQuery.each({
2488        appendTo: "append",
2489        prependTo: "prepend",
2490        insertBefore: "before",
2491        insertAfter: "after",
2492        replaceAll: "replaceWith"
2493    },
2494
2495    //each函数给callback函数(也就是下面这个函数)传入属性的名称(name),以及属性的值(original)
2496    //为了让更好地说明代码的思路,我结合一个具体的值来讲解.就假设我们当前处理的是appendTo(它的值是'append')
2497    //那么name就是'appendTo',original就是'append'.
2498    function(name, original){
2499
2500
         //这里的name如果是'appendTo',那么我们就是在定义一个jQuery.fn.appendTo函数
2501        jQuery.fn[ name ] = function() {
2502
         //记录下传给这个函数的所有参数,我们假设args的值是['"<b>Hello</b>"']
2503            var args = arguments;
2504
2505            //注意,在这个scope(作用范围)内的this指的是一个jQuery对象.
2506
2507            //each函数处理过后,依然返回这个jQuery对象的引用.
2508            //那each到底处理的内容是什么呢?继续往下看:
2509            return this.each(function(){
2510
2511
         //注意,在这个scope(作用范围)内的this指的则是,jQuery对象内匹配元素集合中的每一个元素.
2512
2513                //对传进来的参数进行遍历.注意我们已经在上面假设args == ['"<b>Hello</b>"']
2514
         //将所有的假设数据套进来看看是什么意思:(只看循环体内的代码)
2515                /*
2516                 * jQuery('<b>Hello</b>')['append'](this);
2517                 */
2518
         //可以看到,我们使用'<b>Hello</b>'新建了一个jQuery对象,然后调用'原来的'append方法,反过来将
2519
         //匹配元素集合中的元素this(注意这里的this不是jQuery对象而是一个普通的DOM元素)加进了它的childNodes中.
2520                for ( var i = 0, length = args.length; i < length; i++ )
2521                    jQuery( args[ i ] )[ original ]( this );
2522            });
2523        };
```

```
});
/*
 *
这里each调用与上面的each调用其目的也是大同小异:让下列方法成为jQuery对象的方法.
 *
 * 这组函数大部分与元素的属性以及样式有关.
 */
jQuery.each({
    /**
     *
清空jQuery对象内匹配元素集合中每一个元素上的name所指定的属性值的内容.
     *
     * @param {string} name 属性名称
     */
    removeAttr: function( name ) {

//注意,在这里的this引用的是匹配元素集合中的每一个元素(它一般是一个DOM元素).在jQuery中想要清楚地知道

//this所引用的对象是什么并不容易,尤其是在跟each函数'搅'在一起的时候.如果你真的被jQuery的this搞糊涂了,
        //那么我建议你先看看each的中文注释.


//为了避免remove掉一个并不存在属性,以下三行代码首先将name所指定的属性置为"",这样不管你有没有name所指定
        //的属性,反正现在你是有了.然后调用removeAttribute来删除该属性.

        jQuery.attr( this, name, "" );

        if (this.nodeType == 1/* Node.ELEMENT_NODE */)
            this.removeAttribute( name );
    },
    /**
     *
为元素添加一个类名.它将会在元素原有的类名后面接上空格,然后接上新的类名.
     *
可以看到,它的内部调用了jQeury.className命名空间内的add函数完成任务.jQuery.className内的函数只是内部
     * 使用的,并不对外公开.可以参考jQuery.className的中文注释.
     *
     * @param {string} classNames
     */
    addClass: function( classNames ) {
        //this引用的是一个普通的DOMElement元素.不是jQuery对象.
        jQuery.className.add( this, classNames );
    },
    /**
     * 为元素移除classNames所指定的类名.具体说明同addClass函数.
```

```
2569        *
2570        * @param {string} classNames 字符串,形如"class1 class2 class3".
2571        */
2572       removeClass: function( classNames ) {
2573           //this引用的是一个普通的DOMElement元素.不是jQuery对象.
2574           jQuery.className.remove( this, classNames );
2575       },
2576
2577       /**
2578        * 交替类名.
2579        *
           其实作用很简单:如果元素有classNames指定的类名就去掉这个类名,如果没有这个类名就添加上这个类名.
2580        * 函数说明类同于addClass.
2581        *
2582        * @param {Object} classNames
2583        */
2584       toggleClass: function( classNames ) {
2585           //this引用的是一个普通的DOMElement元素.不是jQuery对象.
2586           jQuery.className[ jQuery.className.has( this, classNames ) ? "remove" : "add" ]( this, classNames );
2587       },
2588
2589       /**
2590        *
           无参的情况下让匹配元素集合中的每一个元素从它的parentNode中移除(remove)出来.
2591        * 有参的话,那么就只把selector所指定的元素移除.
2592        *
2593        * @param {Object} selector
2594        */
2595       remove: function( selector ) {
2596
2597           //this引用的是一个普通的DOMElement元素.不是jQuery对象.
2598
2599           /* 有两种情况就会执行下面的if语句:
2600            * (1) 没有传入selector.
               jQuery会将这种情况会默认为'*',即选择所有的元素.
2601            * (2)
               有传入selector,但是这个selecotr并不能选择到任何的元素.即下面代码中的r.length == 0
2602            *
2603            *
               注意,jQuery.filter函数将返回一个对象,其内有一个名为r的属性.这个属性装着的就是filter执行后的结果集.
2604            *
               它是一个数组.因此通过查询这个集合的length属性就能够判断selector到底有没有选择到元素.具体请参考
2605            * jQuery.filter的中文注释.
2606            */
2607           if ( !selector || jQuery.filter( selector, [ this ] ).r.length ) {
2608
2609               // Prevent memory leaks
2610               //
               翻译:防止内存泄漏.以下三行代码的目的是在删除节点之前将节点对应的所有'遗物'删除掉.这些包括元素的事件
2611               //
```

```
监听器和它缓存的数据.如果不做这个工作,那么将会有可能导致内存泄漏.
2612                jQuery( "*", this ).add(this).each(function(){//
<-用this的所有后代节点新建一个jQuery对象
2613                                                                //
然后用add方法将自己也加进这个jQuery对象的
2614                                                                //
匹配元素集合中.最后就对集合中的所有元素都
2615                                                                //
移除事件监听器和缓存数据.
2616                    jQuery.event.remove(this);
2617                    jQuery.removeData(this);
2618                });
2619
2620                //叫元素的双亲节点删除元素自己...
2621                if (this.parentNode)
2622                    this.parentNode.removeChild( this );
2623            }
2624        },
2625
2626        /**
2627         * 删除匹配元素集合中所有的子节点。
2628         */
2629        empty: function() {
2630
2631            //注意,this引用的是一个普通的DOMElement元素.不是jQuery对象.
2632
2633            // Remove element nodes and prevent memory leaks
2634            //
其实单就清除子节点这个功能来说,用下面那个while循环已经能完事.但是由于r
emove函数会做一些防止内存泄漏的
2635            //
措施.故我们在while之前调用remove先清理一次.再调用while循环来善后.(请参
考jQuery.remove的中文注释)
2636            jQuery( ">*", this ).remove();
//">*"的意思是说,this下的所有后代节点.
2637
2638            // Remove any remaining nodes
2639            while ( this.firstChild )
2640                this.removeChild( this.firstChild );
2641        }
2642    },
2643    //这个就是each所调用的callback函数.它将上述每一个函数包装一下,然后把他
们'接'到每一个jQuery对象上.至于'包装'的内容
2644    //是什么,请继续往下欣赏:
2645    function(name, fn){
2646        jQuery.fn[ name ] = function(){
2647
//调用each函数在匹配元素集合中的每一个元素上都调用一遍那个函数(即fn).
2648            return this.each( fn, arguments );
2649
2650            //如果还是不明白each的作用,建议参考jQuery.each函数的中文注释.
2651        };
2652    });
2653
2654
2655    /**
2656     * 为jQuery对象添加两个函数:width函数和height函数.
2657     * 他们分别用来获取/设置jQuery对象内匹配元素集合中的宽度和高度.
```

```
2658     *
2659     *
         注意,匹配元素集合内的元素大部份情况下是普通的HTMLElement元素,但是也有可能是window或者是document对象
2660     *
         要处理这种情况,随之而来要处理就是大量的在window和在document上的不兼容问题.从而使得本函数显得有些复杂.
2661     */
2662    jQuery.each([ "Height", "Width" ], function(i, name){
2663        var type = name.toLowerCase();//变成小写
2664
2665        //为jQuery对象添加width和height方法.
2666        jQuery.fn[ type ] = function( size ) {
2667
2668            //
         这个函数里面用到了嵌套的?:运算符,所以要小心区分哪个':'是属于那个'?'的.
2669
2670            //
         COMP:在这里,不同浏览器之间关于窗口和文档大小的兼容性问题的解决比较精彩,值得学习!
2671            //
         同时也值得注意,当本函数在获取window的width/height时,实际上获取的是客户区的大小.也即经常听说的
2672            // '视口'的大小,而并不是整个程序窗口的大小.
2673
2674            // Get window width or height
2675            return this[0] == window ?
2676                // Opera reports document.body.client[Width/Height]
         properly in both quirks and standards
2677                jQuery.browser.opera && document.body[ "client" + name ]
         ||
2678
2679                // Safari reports inner[Width/Height] just fine (Mozilla
         and Opera include scroll bar widths)
2680                jQuery.browser.safari && window[ "inner" + name ] ||
2681
2682                // Everyone else use document.documentElement or
         document.body depending on Quirks vs Standards mode
2683                /* 教学时间:
2684                 *
         实际上,以下这行代码主要是面向IE浏览器.在IE怪癖模式下或者在IE的标准模式下视口的大小来源是不同的.在
2685                 *
         没有DOCTYPE声明的页面中,视口大小(clientWidth/clientHeight)在doucment.body上.而当有了DOCTYPE
2686                 *
         声明之后,他们就在document.doucmentElement上了.实际上,w3c把视口大小的名称定为innerWidth/Height
2687                 *
         而IE以及Opera等则把它命名为clientWidth/Height.为了迎合IE系的开发者,Firefox等浏览器都实现了
2688                 * documentElement.其使用方法也是一样的.
2689                 */
2690                document.compatMode == "CSS1Compat" && document.
         documentElement[ "client" + name ] || document.body[ "client" + name
         ] :
2691
2692                // Get document width or height
```

```
2693            this[0] == document ?
2694                // Either scroll[Width/Height] or
    offset[Width/Height], whichever is greater
2695                Math.max(
2696                    /* 各款浏览器在对待scrollWidth/scrollHeight
    offsetWidth/offsetHeight上表现迥异
2697                     * TODO:姑且先记住以下代码是一种解决方案吧...
2698                     */
2699                    Math.max(document.body["scroll" + name], document
    .documentElement["scroll" + name]),
2700                    Math.max(document.body["offset" + name], document
    .documentElement["offset" + name])
2701                ) :
2702
2703                // Get or set width or height on the element
2704                // 翻译:获取/设置 元素的width/height
2705                size == undefined ?
    //如果没有给函数传进值来,那就表示要获取值,否则就是设置值
2706                    // Get width or height on the element
2707                    // 翻译:获取元素的width/height
2708                    (this.length ? jQuery.css( this[0], type ) : null
    ) :
2709
2710                    // Set the width or height on the element
    (default to pixels if value is unitless)
2711                    // 翻译:设置元素的width 或者
    height(如果没有带单位,缺省使用px)
2712                    //
    注意,这里调用了jQuery对象的css方法而不是jQuery.css方法.jQuery.css方法
    只能用来返回元素
2713                    //
    的样式值,是不能用来设置样式值的.而这里的this.css是一个jQeury对象实例的
    方法.这两者是不一样的
2714                    //
    请具体参考jQuery.fn.css方法和jQuery.fn.attr方法的中文注释.
2715                    this.css( type, size.constructor == String ? size
     : size + "px" );
2716        };
2717    });
2718
2719    // Helper function used by the dimensions and offset modules
2720    // 翻译:在尺寸模块和offset模块会用到的辅助函数.
2721    // 注意, "dimensions and offset
    modules"是jQuery源代码中的最后一个模块.
2722    // 本函数使用jQuery.curCSS函数来获取元素某个样式的值(数字部分).
2723    function num(elem, prop) {
2724        return elem[0] && parseInt( jQuery.curCSS(elem[0], prop, true),
    10 ) || 0;
2725    }
2726
2727    //COMP:
2728    //COMP:TODO:Safari的问题.
2729    var chars = jQuery.browser.safari && parseInt(jQuery.browser.version)
     < 417 ?
2730            "(?:[\\w*_-]|\\\\.)" :
2731            "(?:[\\w\u0128-\uFFFF*_-]|\\\\.)",
2732        quickChild = new RegExp("^>\\s*(" + chars + "+)"),
2733        quickID = new RegExp("^(" + chars + "+)(#)(" + chars + "+)"),
```

```
2734        quickClass = new RegExp("^([#.]?)(" + chars + "*)");
2735
2736    /**
2737     *
        这里将通过extned函数,扩展jQuery的功能.这些功能主要与正则表达式,过滤器有关.由于涉及正则表达式,看起来会比较吃力.
2738     */
2739    jQuery.extend({
2740        /**
2741         * 一个问题:类似":last" 这样的伪类选择器是怎样被选择出来的?
2742         * 下面的这个 jQuery.expr 对象内的函数将会告诉你答案.
2743         *
2744         *
        原来在我们的表达式最终将会被jQuery.filter函数接收到,然后在filter函数通过正则表达式(parse数组内的正则表达式. parse数组?查找一下吧)
2745         *
        将选择器中的符号和名称部分截取出来(如':last'的符号部分就是':',名称部分
2746         *
        就是'last').查看符号以及名称是什么,然后对号入座找到相应的函数进行处理.如我们的':last'就会
2747         *
        由jQuery.expr[':']['last']函数进行处理.又如':first'将会由.jQuery.expr[':'].first函数进行处理.
2748         *
2749         *
        你可以看到,这些函数并没有做什么,只是根据自己的条件,做出最后的布尔判断而已.实际上,这些函数最后将会传给jQuery.grep
2750         *
        函数并作为它(jQuery.grep)的一个过滤函数使用.如果下面列出的任意一个的函数(如first)返回false,意思就是告诉grep
2751         *
        函数当前处理的这个元素不符合过滤的要求,当前元素不需加入最后的结果集.如果返回true,则说明当前元素符合过滤的条件,可
2752         * 以把它放入最后的结果集.
2753         *
2754         *
        最后注意,jQuery.grep函数有一个布尔类型的参数(第三个参数),它将用来控制过滤函数返回true或false时的具体含义.
2755         *
        ture是要留下这个元素呢,还是不要?这都由这个布尔参数来决定.默认情况下(不传入这个参数时)就是我上面所说的过滤逻辑
2756         * 即false不要,true留下.
2757         *
2758         * 建议再查看jQuery.filter函数之前,查看jQuery.grep函数的中文注释.
2759         */
2760        expr: {
2761            "" : function(a,i,m){return m[2]=="*"||jQuery.nodeName(a,m[2]);},
2762            "#" : function(a,i,m){return a.getAttribute("id")==m[2];},
2763            ":" : {
2764                /*
                ':'命名空间下的以下这些函数的作用可以查阅这份jQuery在线Api文档:
2765                 *
                http://jquery-api-zh-cn.googlecode.com/svn/trunk/index.html的'选择器'部分
2766                 *
2767                 * 我在这里就不copy了,那么我在这里说些有技术含量的——
2768                 *
```

```
以下这些函数在什么情况下、如何被使用到呢?以下举个例子说明:
2769                 *
当我们使用$('p:first')创建一个jQuery对象的时候,':first'是这样被处理的:
2770                 *
jQuery对象首先用'p'选择出document中的所有p放入到匹配元素集合中去.然后调用jQuery.fn.find
2771                 *
函数进行过滤.find函数又调用jQuery.grep函数(它是一个静态函数)进行过滤,jQuery.grep函数又调用
2772                 *
jQuery.find(它是一个静态函数)进行过滤,jQuery.find函数又调用filter函数进行过滤.这时候,
2773                 *
jQuery.filter函数就从selector中截取出':first'(注意,selector是被一层一层传进来的).jQuery.filter
2774                 *
函数首先就用':'和'first'在jQuery.expr中查找合适的过滤函数,最后,它找到了jQuery.expr[':'].first
2775                 *
那么jQuery.filter最后就把这个函数交给jQuery.grep做最后的过滤,最后得到过滤的结果集.
2776                 *
2777                 *
是不是感觉很复杂?如果你认为这个并不重要,那你仅仅是需要知道jQuery.filter和jQuery.grep
2778                 * 函数在选择结果筛选方面处于核心地位就可以了.
2779                 *
2780                 *
最后要注意,这些函数每个参数的含义:a表示当前处理的匹配元素集合中的那个元素;i表示这个元素在匹配元素集
2781                 *
合中的索引,m就是在jQuery.filter函数中对选择器进行正则匹配的结果集,它是一个数组.具体见jQuery.filter
2782                 * 函数.
2783                 */
2784                // Position Checks
2785                lt: function(a,i,m){return i<m[3]-0;},
2786                gt: function(a,i,m){return i>m[3]-0;},
2787                nth: function(a,i,m){return m[3]-0==i;},
2788                eq: function(a,i,m){return m[3]-0==i;},
2789                first: function(a,i){return i==0;},
2790                last: function(a,i,m,r){return i==r.length-1;},
2791                even: function(a,i){return i%2==0;},
2792                odd: function(a,i){return i%2;},
2793
2794                // Child Checks
2795                "first-child": function(a){return a.parentNode.getElementsByTagName("*")[0]==a;},
2796                "last-child": function(a){return jQuery.nth(a.parentNode.lastChild,1,"previousSibling")==a;},
2797                "only-child": function(a){return !jQuery.nth(a.parentNode.lastChild,2,"previousSibling");},
2798
2799                // Parent Checks
2800                parent : function(a){return a.firstChild;},
2801                empty : function(a){return !a.firstChild;},
2802
2803                // Text Check
```

```
2804            contains: function(a,i,m){return (a.textContent||a.innerText||jQuery(a).text()||"").indexOf(m[3])>=0;},
2805
2806            // Visibility
2807            visible: function(a){return "hidden"!=a.type&&jQuery.css(a,"display")!="none"&&jQuery.css(a,"visibility")!="hidden";},
2808            hidden: function(a){return "hidden"==a.type||jQuery.css(a,"display")=="none"||jQuery.css(a,"visibility")=="hidden";},
2809
2810            // Form attributes
2811            enabled: function(a){return !a.disabled;},
2812            disabled: function(a){return a.disabled;},
2813            checked: function(a){return a.checked;},
2814            selected: function(a){return a.selected||jQuery.attr(a,"selected");},
2815
2816            // Form elements
2817            text: function(a){return "text"==a.type;},
2818            radio: function(a){return "radio"==a.type;},
2819            checkbox: function(a){return "checkbox"==a.type;},
2820            file: function(a){return "file"==a.type;},
2821            password: function(a){return "password"==a.type;},
2822            submit: function(a){return "submit"==a.type;},
2823            image: function(a){return "image"==a.type;},
2824            reset: function(a){return "reset"==a.type;},
2825            button: function(a){return "button"==a.type||jQuery.nodeName(a,"button");},
2826            input: function(a){return /input|select|textarea|button/i.test(a.nodeName);},
2827
2828            // :has()
2829            has: function(a,i,m){return jQuery.find(m[3],a).length;},
2830
2831            // :header
2832            header: function(a){return /h\d/i.test(a.nodeName);},
2833
2834            // :animated
2835            animated: function(a){return jQuery.grep(jQuery.timers,function(fn){return a==fn.elem;}).length;}
2836        }
2837    },
2838
2839    // The regular expressions that power the parsing engine
2840    // 翻译:让解析引擎具有无比威力的正则表达式
2841    // 这些正则表达式在filter函数中被使用.传进给filter函数的选择器将会经过这些正则表达式的测试.
2842    // 具体看jQuery.filter函数.
2843    parse: [
2844        // Match: [@value='test'], [@foo]
2845        /^(\[) *@?([\w-]+) *([!*$^~=]*) *('?"?)(.*?)\4 *\]/,
2846
2847        // Match: :contains('foo')
2848        /^(:)([\w-]+)\("?'?(.*?(\(.*?\))?[^(]*?)"?'?\)/,
2849
2850        // Match: :even, :last-child, #id, .class
2851        new RegExp("^([:.#]*)(" + chars + "+)")
2852    ],
```

```
2853
2854        //"
2855        /**
2856         *
    一个能够处理多个选择器的过滤器.可以看到multifilter其实是调用jQuery.filter函数来实现功能的.其'处理多个
2857         * 选择器'的实质是多次调用filter函数.
2858         *
2859         * 什么是多个选择器?如"form > input,
    #id"内就有两个过滤器,他们之间用','隔开.
2860         *
2861         * expr   选择器,字符串.
2862         * elems  即选择器其作用的范围.也就是通常所说的'上下文'.
2863         * not
    一个开关,表示是否开启"非模式".到底是'过滤掉(非模式)'还是'过滤剩'呢?这都由not来决定.not为true表示,选择器t所指定
2864         *
    的元素全部从过滤结果中去掉.而not为false或者不传入时,则表示选择器t所指定的元素全部加进结果集中.
2865         */
2866        multiFilter: function( expr, elems, not ) {
2867            var old,
    //它用来装传进来的正则表达式.由于expr在下面的while中会不断地被while修改,old就用来记录expr
2868                    //在这一次修改前前的状态.以通过 expr ==
    old来判断expr是否被修改过.
2869                cur = [];//当前经过过滤的结果集.
2870
2871                            //循环条件加上expr !=
    old是为了应对expr没有被匹配到的情况.在这种情况之下,while就不会停了
2872            while ( expr && expr != old ) {
2873                old = expr;
    //保存当前expr的值,等到下一次循环时看看expr有没有被改变.
2874
    //调用jQuery.filter进行真正的过滤.下面对jQuery.filter函数的返回值进行说明:
2875
    //f是一个对象,它有两个属性,分别是r和t。r是过滤后的结果集合,而t就是经过filter截取之后的expr.
2876                //如我们将'form > input,
    #id'传给jQuery.filter函数.经过处理之后,这个字符串将会变成',#id'并存放在
2877
    //一个对象的t属性当中,然后这个对象被返回并给f接收了.于是现在的f.t就是',#id'了.
2878                var f = jQuery.filter( expr, elems, not );
2879
    //把f.t内位于字符串前端的','去掉.如把',#id'的','去掉就成为了'#id'.
2880                expr = f.t.replace(/^\s*,\s*/, "" );
2881
2882                cur = not ? //
    是否有开启'非模式'?如果有就把f.r赋给elems留作下一次过滤的上下文.即在上一次过滤掉的结果
2883                        elems = f.r : //上再过滤掉后面的选择器所指定的元素.
2884                        jQuery.merge( cur, f.r );
    //如果没有开启'非模式',那么就把所有过滤器过滤的结果集统统合并在一起.
2885
2886                //
```

```
            经过了jQuery.filter这一回合的处理,expr会被剪掉已经匹配的那一部分选择器了.
2887            //
            如果被剪掉之后expr还没有空,证明还有活干,还要继续匹配,这样循环就继续,一直到expr空了或者expr==old为止.
2888        }
2889
2890        return cur;//最后把结果集返回.
2891    },
2892
2893    /**
2894     * 在context中搜寻t所指定的子元素,并将结果放在一个数组中返回.
2895     *
2896     * @param {string} t - 选择器
2897     * @param {Object} context -
            选择器进行选择所依赖的上下文.可以看到如果没有传入,那它就是document.
2898     */
2899    find: function( t, context ) {
2900        // Quickly handle non-string expressions
2901        // 如果选择器不是string,那就直接把t装在一个数组中返回.
2902        if ( typeof t != "string" )
2903            return [ t ];
2904
2905        // check to make sure context is a DOM element or a document
2906        //
            保证context至少是一个DOM元素(HTMLElement或者HTMLDocument),如果都不是,那就返回一个空集合.
2907        if ( context && context.nodeType != 1/* Node.ELEMENT_NODE */
            && context.nodeType != 9/* Node.DOCUMENT_NODE */)
2908            return [ ];
2909
2910        // Set the correct context (if none is provided)
2911        // 至少让这个context为document元素: 有传入context当然好,
            没的话就让context为document
2912        context = context || document;
2913
2914        // Initialize the search
2915        // 初始化搜寻
2916        var ret = [context], //最终的结果集
2917            done = [],
2918            last, //这个在下面的注释中有讲解.
2919            nodeName;
            //它用来装待会在t中匹配出来的后代节点的节点类型名称.
2920
2921        // Continue while a selector expression exists, and while
2922        // we're no longer looping upon ourselves
2923        /* 在以下的while循环中,将会对t(提个醒,
            t是选择器字符串)进行裁减.每经过一次循环,t就有可能被裁减一次.
2924         *
            如果经过一次循环回来后发现t已经是''或者t跟本次循环开始前是一样的(last != t),那么循环就终止.
2925         * 这个while循环的目的在于遍历t内的所有选择器.
2926         */
2927        while ( t && last != t ) {
2928            var r = [];//结果集合初始化为空集合.
2929            last = t;
            //让last保存当前的t的内容,因为在等下的处理当中t的内容很可能被更改.并且我们通过对比t前后的内容以判断t是否得到匹配.
```

```
2930
2931            t = jQuery.trim(t);//去掉t两头空格
2932
2933            var foundToken = false,
    //一个标志,告诉我们整个查找是否已经完成了.
2934
2935            // An attempt at speeding up child selectors that
2936            // point to a specific element tag
2937            //
    原文翻译:加快指向特定标签的子选择器的速度...(有点不知所云,是吧...别急,
    继续往下看)
2938                re = quickChild,
    //quickChild这个正则表达式已经在整个正则过滤模块开始的时候已经定义.可
    以使用Ctrl+F倒回去看看
2939                                        //类似'>
    span'这样的选择器就叫做quickChild.
2940                m = re.exec(t);
    //在t中查找符合re这个正则表达式所描述的字串,
    并将匹配的结果装入一个数组返回给m
2941
2942                //如果 exec 方法没有找到匹配，则它返回
    null。如果它找到匹配，则 exec 方法返回一个数组，并且更新全局 RegExp
    对象的属性，以
2943
    //反映匹配结果。数组的0元素包含了完整的匹配，而第1到n元素中包含的是匹
    配中出现的任意一个子匹配。即下标1到n表示正则表达式的分组号
2944                //如果你还不是太清楚,别急,继续往下看.
2945
2946            // 如果有匹配的字串,说明t是类似"> p
    span"的情况,那就首先从context的childNodes里面找,看看有没有符合要求的子
    节点
2947            if ( m ) {
2948                                //
    m[1]表示第一个分组(quickChild的第一个分组),如果t为"form >
    input",那么m[0] == "> input", m[1] == "input"
2949                nodeName = m[1].toUpperCase();
2950
2951                // Perform our own iteration and filter
2952
2953                //接下来的for循环是遍历context的childNodes,
    把每一个tagName是nodeName的加进结果集来.如果nodeName是*,那就把所有的ch
    ildNodes加进结果
2954                //集来
2955
2956                                        //注意,ret已经初始化为ret[context]
2957                for ( var i = 0; ret[i]; i++ )
2958                    for ( var c = ret[i].firstChild; c; c = c.
    nextSibling )
2959                                                // nodeName = m[1]
2960                        if ( c.nodeType == 1/* Node.ELEMENT_NODE */
    && (nodeName == "*" || c.nodeName.toUpperCase() == nodeName) )
2961                            r.push( c );//把节点加进结果集
2962
2963                ret = r;
2964                t = t.replace( re, "" );
    //把匹配的字符去掉,说明这次匹配已经完成
2965
2966
```

```
//接下来看t(提醒,它是一个selector)里面是不是还有空格,
如果有,说明是活还没干完,t内仍然有选择器. 那继续在找到的集
2967                    //合里面找"儿子的儿子",继续循环
2968                    if ( t.indexOf(" ") == 0 ) continue;
2969                    foundToken = true;//做一个记号.说明查找完成了.
2970                }//end
if.利用这个if来判断t不是不是符合quickChild的字符模式.
2971
2972                // else是说,
t这个选择器里面并没有quickChild模式的字串,即没有类似"> input"这样的串
2973                else {
2974                    //再来定义一个正则表达式
2975                    re = /^([>+~])\s*(\w*)/i;
2976
2977                    // 如果跟t匹配的话...(如t的值为:"> input" 或者 "+
input" 又或者 "~ input")
2978                    if ( (m = re.exec(t)) != null ) {
2979                        r = [];//结果集合初始化为空集合.
2980
2981                        var merge = {};
2982                        nodeName = m[2].toUpperCase();
//注意啊,在这里的分组2才是nodeName!这个跟quickChild不一样啊.
2983                        m = m[1];// m[1]的就是">" "+" "~" 中的一种
2984
2985
//注意,for循环中的ret被初始化为[context].这个for的意思就是在所有的cont
ext中查找t所指定的子元素.
2986                        for ( var j = 0, rl = ret.length; j < rl; j++ ) {
2987                            // m如果是"~"和"+"中的一个, 就让n =
ret[j].nextSibling
2988                            // 否则(m == ">"), 就让n = ret[j].firstChild
2989                            //
这是因为'>'表示的是一种父子的关系,而'~'和'+'则表示一种兄弟的关系.可以
参考这篇在线文档:
2990
//http://jquery-api-zh-cn.googlecode.com/svn/trunk/index.html
的'选择器'->'层级'
2991                            var n = m == "~" || m == "+" ? ret[j].
nextSibling : ret[j].firstChild;
2992
2993
2994                            for ( ; n; n = n.nextSibling )
2995                                if ( n.nodeType == 1/* Node.ELEMENT_NODE
*/ ) {//只对元素节点感兴趣
2996                                    var id = jQuery.data(n);
//jQuery.data函数原来的作用是获取存在这个节点上的数据.
2997
//但是当我们只传入第一个元素给它的时候,它只返回元素在全局数据缓存区中
的缓存区id.
2998
//这个id由jQuery全局同一分配,如果一个元素并没有id的时候,jQuery.data会
给它
2999
//分配一个.这里获取这个id仅仅是为了要做一个记号,防止重复合并同一个元素
而已
3000
//建议参考jQuery.data
3001
```

```
3002
                //merge初始是一个空对象
3003                                if ( m == "~" && merge[id] )
3004                                    break;
                //如果m=="~"即要选择所有的siblings &&
                这个节点(n)已经被加入了结果集,马上跳出循环
3005
3006
3007
3008
                //如果没有指定nodeName(那就是说不管什么元素,统统都要啦)或者n的nodeName
                就是指定nodeName,那就把元素加进结果集
3009                                if (!nodeName || n.nodeName.
                toUpperCase() == nodeName ) {
3010                                    if ( m == "~" ) merge[id] = true;
                //作个记号,说明这个元素已经被加进结果集了.
3011                                    r.push( n );//把元素加进结果集
3012                                }
3013
3014                                if ( m == "+" ) break;
                //如果m=="+",只要一个就够了.如'label +
                input'就表示紧接着label的那个input
3015
                //另外还需要注意,在下一次循环中,可能还有元素要加进结果集合.因为'紧接着
                label的那
3016
                //个input'并不能指定一个唯一的元素.
3017                            }
3018                    }//end for
3019
3020                    ret = r;//将r中的结果交给ret,ret是最后的结果集.
3021
3022                    // And remove the token
3023                    t = jQuery.trim( t.replace( re, "" ) );
                //把匹配的字符串去掉,说明这个匹配已经完成
3024                    foundToken = true;
                //作个记号,说明还是有收获的,选择器匹配到了元素.
3025                }
3026            }
3027
3028            /*
                上面的代码目的是想看看t是否匹配quickChild的模式.而下面的的一个if就是检
                查匹配的结果.如果下面的if内的代码能够执行
3029             *
                这表明t并不是quickChild所描述的那种字符串模式.那么我们就会用其他的字符
                模式去尝试匹配它.请继续观看...
3030             */
3031
3032            // See if there's still an expression, and that we
                haven't already
3033            // matched a token
3034            // t里面是不是还有selector &&
                还没找到合适的元素?(!foundToken)
3035            if ( t && !foundToken ) {
3036
3037                // Handle multiple expressions
3038                //
                如果现在的t里面不含有",",说明t是一个单独的选择器.注意','是用来分隔多个
```

```
选择器的.如'#id1,#id2'就是两个选择器.
3039                if ( !t.indexOf(",") ) {
3040                    // Clean the result set
3041
//如果ret[0]还是context,说明前面的操作并没有找到合适的元素,把context给"弹"出来,清空ret
3042                    if ( context == ret[0] ) ret.shift();
3043
3044                    // Merge the result sets
3045                    done = jQuery.merge( done, ret );
//done才是最后要返回的结果数组.而ret,r都是临时的结果集.
3046
3047                    // Reset the context
3048                    r = ret = [context];
3049
3050                    // Touch up the selector string
3051                    t = " " + t.substr(1,t.length);
3052
3053                }
3054
3055                //
如果现在的t里面含有",",说明t是一个'复合选择器(multiSelector)',即t内还有多个选择器(它们之间用','隔开)
3056                else {
3057                    // Optimize for the case nodeName#idName
3058                    var re2 = quickID;
//quickID这个正则表达式描述的是nodeName#idName这样的字符串模式
3059                    var m = re2.exec(t);
//把匹配的结果装进一个数组,然后返回给m
3060
3061                    // Re-organize the results, so that they're consistent
3062                    //
重新组织一下匹配的结果,这么做是为了和上面的处理保持一致:m[1]应该是一个符号,而m[2]就是符号的右边的名称
3063                    if ( m ) {
3064                        m = [ 0, m[2], m[3], m[1] ];
3065                    }
3066                    // 如果t还不能得到匹配,
那现在的这个t内可能是含有传统的id或class(如'#id','.className'),用quickClass
3067                    //
正则式去尝试匹配它.注意别被quickClass的名字'骗'了.通过查看它的代码可以清楚看到,quickClass即匹配id,又匹配className
3068                    else {
3069                        // Otherwise, do a traditional filter check for
3070                        // ID, class, and element selectors
3071                        re2 = quickClass;
//quickClass正则表达式描述的是类似"#id",".className"这样的字符串模式
3072                        m = re2.exec(t);
//测试t的模式,然后把匹配的结果放到放到一个数组里面,然后返回给m
3073                    }
3074
3075                    // 将id名或者是class名内的"\"去掉
3076                    m[2] = m[2].replace(/\\/g, "");
3077
3078                    var elem = ret[ret.length-1];
```

```
3079
3080                    // Try to do a global search by ID, where we can
3081                    //
如果m[1]是'#',那我们就看看当前的文档是不是xml文档.如果不是xml文档,并且elem具有getElementById,
3082                    //
那就表明elem就是一个HTMLDocument元素.注意,getElementById并不能用在XML文档当中.
3083                    if ( m[1] == "#" && elem && elem.getElementById && !jQuery.isXMLDoc(elem) ) {
3084                        // Optimization for HTML document case
3085                        //
程序能运行到这里,说明elem是一个HTMLDocument元素.直接使用它的getElmentById函数,m[2]是匹配到的id名
3086                        var oid = elem.getElementById(m[2]);
3087
3088                        // Do a quick check for the existence of the actual ID attribute
3089                        // to avoid selecting by the name attribute in IE
3090                        // also check to insure id is a string to avoid selecting an element with the name of 'id' inside a form
3091                        /*
COMP:在IE中getElementById可能会把name为同样值的元素给选择出来,如:
3092                         * <form>
3093                         *      <input name="goodInput"></input>
3094                         * </form>
3095                         *
当我们使用document.getElmentById('goodInput')的时候,就会把这个鬼input选择进来.
3096                         *
所以为了避免这种情况给我们带来的乱子,我们首先要检测一下当前浏览器是不是IE或者Opera(他们都有这个问题),然后再看看他们的id
3097                         *
属性是不是我们所要的那个id.如果不是,那么调用代码
Query('[@id="'+m[2]+'"]',elem)[0] 来获得一个真正的,id为我们所指
3098                         * 定的值的元素,然后让oid指向这个元素.
3099                         */
3100                        if ( (jQuery.browser.msie||jQuery.browser.opera) && oid && typeof oid.id == "string" && oid.id != m[2] )
3101                            oid = jQuery('[@id="'+m[2]+'"]', elem)[0];
3102
3103
3104                        // Do a quick check for node name (where applicable) so
3105                        // that div#foo searches will be really fast
3106                        /*
在上面的注释中讲过,像'div#foo'这样的字符串模式被quickID匹配之后,其结果数组被保存到了m中.然后这个m经过了一个小小的
3107                         *
调整之后,m内的元素顺序发生了改变.以'div#foo'为例,这个时候m[1]=='#',m[2]=='foo',m[3]=='div',所以作者才会说
3108                         * "Do a quick check for node name...".
3109                         *
经过m[3]nodeName的测试,证实oid的确具有m[3]所指定的nodeName,那就把oid装进数组中并作为结果返回.如果oid根本就没有
3110                         * m[3]所指定的nodeName,则返回一个空的数组.
3111                         */
```

```
3112                            ret = r = oid && (!m[3] || jQuery.nodeName(
    oid, m[3])) ? [oid] : [];
3113                        }
3114                        //如果不是上面那种情况
3115                        else {
3116                            // We need to find all descendant elements
3117                            // 翻译:我们需要找出所有的后代元素
3118                            //
    来一个for循环,遍历初始化时装入ret中上下文元素.也就是说,现在我们不是要
    找出所有的后代元素吗?那到底要找出谁的后代呢?
3119                            // 答案就是,他们都在ret中.
3120                            for ( var i = 0; ret[i]; i++ ) {
3121                                // Grab the tag name being searched for
3122                                var tag = m[1] == "#" && m[3] ? m[3] : m[
    1] != "" || m[0] == "" ? "*" : m[2];
3123
3124                                // Handle IE7 being really dumb about
    <object>s
3125                                // COMP:TODO:在IE
    7中,quickID这个正则表达式似乎不能够正确地工作在<object>元素上.
3126                                //
    tag的匹配结果如果是'*',那并不一定是一个正常的结果.有可能是IE7的<object
    >bug所造成
3127                                //
    下面的这个if语句就是要处理IE7关于'object'的这个问题.
3128                                if ( tag == "*" && ret[i].nodeName.
    toLowerCase() == "object" )
3129                                    tag = "param";
3130
3131                                /*
    getElementByTagName是定义在HTMLElement接口中方法.所有的DOM元素都实现了
    这个接口
3132                                 *
    可以看到,以下代码使用getElementByTagName取得ret中第i个上下文元素的所有
    后代节点,然后把这些后代并入r中.
3133                                 */
3134                                r = jQuery.merge( r, ret[i].
    getElementsByTagName( tag ));
3135                            }
3136
3137                            // It's faster to filter by class and be
    done with it
3138                            // 如果传入的是一个类选择器.
3139                            if ( m[1] == "." )
3140
    //调用类选择器专用的filter函数.它的参数的意思是说,在r中过滤剩下具有m[2
    ]所指定的类名的元素
3141                                r = jQuery.classFilter( r, m[2] );
3142
3143                            // Same with ID filtering
3144                            // ID选择器
3145                            if ( m[1] == "#" ) {
3146                                var tmp = [];
3147
3148                                // Try to find the element with the ID
3149                                //
    只要一找到拥有指定id的元素就可以停止查找,并返回结果了.
3150                                for ( var i = 0; r[i]; i++ )
```

```
3151                                    if ( r[i].getAttribute("id") == m[2]
    ) {
3152                                        tmp = [ r[i] ];
3153                                        break;
3154                                    }
3155
3156                                r = tmp;
3157                            }
3158                            ret = r;
3159                        }//endl else
3160
3161                        t = t.replace( re2, "" );
    //完成对re2的匹配,可以将它从t中取出掉了.
3162                    }//end else.
3163                }
3164
3165                //
    经过上面这么多层的过滤之后(每次过滤都会将t内的匹配字符串变为""),t内如
    果还有存在selector,说明我们'quickXXXX'的选择器
3166                //
    并不能满足需求.那就调用最基本,最原始jQuery.filter过滤器来处理,因为它能
    够处理任何合法形式的选择器.
3167
3168                // If a selector string still exists
3169                if ( t ) {
3170                    // Attempt to filter it
3171                    var val = jQuery.filter(t,r);
    //t是选择器,r是选择器其作用的上下文,也即t的作用范围.
3172                    ret = r = val.r;//val.r是filter过滤后的结果.
3173                    t = jQuery.trim(val.t);
    //val.t则是经过filter处理后,裁减过的选择器.
3174                }
3175
3176            }//end while
3177
3178            // An error occurred with the selector;
3179            // just return an empty set instead
3180            //
    如果传入一个不合法的选择器,即t的值不是正常的值.那么程序运行到这里t是有
    可能还是有值的.在这种异常的情况之下,我们返回一个空的数组作为回应.
3181            if ( t )
3182                ret = [];
3183
3184            // Remove the root context
3185            // 移除根元素
3186            if ( ret && context == ret[0] )
3187                ret.shift();
3188
3189            // And combine the results
3190            done = jQuery.merge( done, ret );
    //done才是最后要返回的结果集.而ret是一个临时的结果集合.
3191
3192            return done;
3193        },
3194
3195        /**
3196         * 从r中过滤掉className不符合要求的元素
3197         * r - 一个result set,
```

```
函数将从这个集合内过滤剩或者过滤掉由m选择器指定的元素
3198         * m - 它是一个style类名.
3199         * not - 是要过滤掉,还是过滤剩呢?由这个参数作为一个开关
3200         */
3201        classFilter: function(r,m,not){
3202            //
给m所指定的类名的左右两边各添加一个空格,是为了防止m所指定的类名被其他
的类名包含的情况,如:
3203            //
'className'就会被'theclassName'包含,从而导致了等下使用indexOf函数判断
的失败.
3204            m = " " + m + " ";
3205            var tmp = [];
3206            //逐个检查r中的每一个元素,看看它是否有m所指定的类名.
3207            for ( var i = 0; r[i]; i++ ) {
3208                var pass = (" " + r[i].className + " ").indexOf( m ) >= 0;
3209                if ( !not && pass || not && !pass )
//这个表达式太精练了,没有办法用语言描述,也很简单...
3210                    tmp.push( r[i] );
3211            }
3212            return tmp;//返回过滤后的结果.
3213        },
3214
3215        /**
3216         *
在r指定的元素里面,过滤出t所指定的元素(t是selector,一般为string类型)
3217         *
3218         * t - 选择器
3219         * r - 选择器执行的上下文.即选择器要在r所指定的元素范围内
3220         * not -
一个开关,表示是否开启"非模式".是'过滤掉'还是'过滤剩'呢?这都由not来决定
.not为true表示,选择器t所指定
3221         *
的元素全部从过滤结果中去掉.而not为false或者不传入时,则表示选择器t所指
定的元素全部加进结果集中.
3222         *
3223         */
3224        filter: function(t,r,not) {
3225            var last;
3226
3227            // Look for common filter expressions
3228            /* t是传进来的一个字符串,它是一个选择器,形如:'selector1
.selector2 :selector3'.(注意这个字符串内有3个
3229             * 选择器)。我们需要逐步截取出每一个选择器.
在'selector1'匹配的结果中再用'.selector2'过滤.一直这么循环
3230             * 下去.while面的这个while循环就是执行上述的功能.
3231             *
3232             *
这里再说一下while循环的循环条件中的last.它的意思是'上一次循环开始时使
用的selector'.由于while的每一
3233             *
次循环中都会对选择器t进行截取,故再每一次循环后t都不应该跟原来的一样.如
果经过循环之后选择器t还是保持原样
3234             * 即 t == last,说明t已经不能再被截取并进行处理了.
3235             */
3236            while ( t && t != last ) {
3237                last = t;
//记录下现在这个seletor的样子,留待下次循环条件检查的时候看看t有没有变
```

```
化.
3238
3239            var p = jQuery.parse, m;
//m等下会用来装正则表达式内的匹配结果.
3240
3241            //p是一个数组,其内装着三个正则表达式,
用来匹配类似这样的字串情形:
3242            //情形一:  [@value='test'], [@foo]
3243            //情形二:  :contains('foo')
3244            //情形三:  :even, :last-child, #id, .class
3245
3246            //注意, p仅要求首次匹配的子串
3247
3248
//遍历上面的三种情况,看看t是当中的哪一种.找出之后,将匹配的结果放入m中,
再做点处理,留待下面使用.
3249            for ( var i = 0; p[i]; i++ ) {
3250                m = p[i].exec( t );
3251
3252                if ( m ) {
//如果m是有值的,说明t内仍然有需要处理的选择器.
3253
3254                    // Remove what we just matched
3255                    // 原文翻译: 把刚才的匹配的字串从t中给去掉
3256                    t = t.substring( m[0].length );
3257
3258                    m[2] = m[2].replace(/\\/g, "");
//就是符号后的字符串,例如匹配的字串为":even",则m[1] == ":" ; m[2] ==
"even"
3259                    break;//找到了t属于那一种情况,那就赶快跳出for吧
3260                }
3261            }
3262
3263            //如果t根本就不匹配, 那就跳出整个while循环,函数返回
3264            if ( !m )
3265                break;
3266
3267
3268            // 好了,找到t属于哪种情况了.
那么对这种情况的每一子情况进行处理
3269            // 下面一连串的if / else if /else 就是分情况进行处理
3270
3271            // :not() is a special case that can be optimized by
3272            // keeping it out of the expression list
3273            // 原文翻译: :not() 是一个特殊的情况.
把它放到表达式之外可以使它得到优化
3274            /* 你一定在骂我'什么狗屎翻译...',我说明一下吧:
3275             *
如果选择器中有':not'则说明将紧跟其后的选择器(即m[3])所匹配的元素从结果
集中过滤出去.这是目标是通
3276             * 过递归掉用filter并传入给它第三个参数(true)达到的.
3277             */
3278            if ( m[1] == ":" && m[2] == "not" )
3279                // optimize if only one selector found (most common
case)
3280                r = isSimple.test( m[3] ) ?  // isSimple =
/^.[^:#\[\.]*$/    , 例解:
m[3]是:not('inner_selector')内的inner_selector
```

```
3281                        jQuery.filter(m[3], r, true).r ://如果就是simple的选择器(以":" 、":"、"."等开头的选择器),那就交回给本函数处理,同时给本函数传多
3282                                                        // 一个标记:not = true. 这个表示要将匹配的元素进行剔除的操作
3283                        jQuery( r ).not( m[3] ); //如果不是simple的选择器比如说不只一个选择器,那就使用更加"专业"的not函数来处理了.
3284
3285                // We can get a big speed boost by filtering by class here
3286                // 原文翻译: 使用class来过滤的话将会使速度得到很大的提升
3287                else if ( m[1] == "." )
                    //如果是使用类来过滤,就调用classFilter来处理
3288                    r = jQuery.classFilter(r, m[2], not);
                    //not起一个开关的作用,为ture时,要的元素不要,不要的元素反而要...
3289
3290                else if ( m[1] == "[" ) {
3291                    var tmp = [], type = m[3];//m[3]是这样的一个值:如整个匹配字符串为"[att $= value]", 那么m[1]=="[" ; m[2]=="attr" ; m[3]=="$="
3292
3293                                                //提醒:r是传进本函数的一个参数.可以看看函数签名(函数头的定义)
3294                    for ( var i = 0, rl = r.length; i < rl; i++ ) {
3295                        var a = r[i],
3296                            z = a[ jQuery.props[m[2]] || m[2] ];
                            //props数组确保m[2]所指定的属性名在JS中是有效的,如dom元素属性class在
3297
                            //JavaScript中应该表述为className
3298
                            //那么z就是a所指代元素的属性值
3299
3300                        //如果属性值是空的 或者属性名称是href/src/selected,你就要把这个属性值"调整"一下
3301                        //为什么要调整一下呢? 因为如果上面的a[jQuery.props[m[2]] || m[2]]返回的值z是null,说明使用快捷方式来获取属性值失败了,那就使用更
3302                        //加"专业"的获取方法:attr
3303
                            //属性的获取在不同的浏览器下会有一些bug,这个需要attr来处理.另外像opacity这样的属性,在IE下在filter内设置,而在w3c浏览器下则是在
3304
                            //style.opacity内设置.这些差异性,都是需要attr来处理的.
3305
                            //多数情况之下,直接获取属性失败都是因为上述的那些差异.
3306                        if ( z == null || /href|src|selected/.test(m[2]) )
3307                            z = jQuery.attr(a,m[2]) || '';
3308
3309
                            //获取到了属性值之后,看看到底选择器要函数干些什么.看看type就知道(m[5]是匹配到的属性值,m[4]是括住这个属性值的单引号或者双引号)
3310                        //
3311                        //"= value"等于value
3312                        //"!= vlaue"不等于value
3313                        //"^= value"以value开头
3314                        //"$= value"以value结尾
3315                        //"*= value"包含value
3316                        if ( (type == "" && !!z ||
```

```
//如果z是有值的(不是undefined或者null),那!!z 就是true,否则为false
                            type == "=" && z == m[5] ||
                            type == "!=" && z != m[5] ||
                            type == "^=" && z && !z.indexOf(m[5]) ||
//!负数 === false
                            type == "$=" && z.substr(z.length - m[5].
length) == m[5] ||
                            (type == "*=" || type == "~=") && z.indexOf(
m[5]) >= 0) ^ not )  //TODO: not很有可能是将集合取非,而不是Resig说的

                          //一个或者多个selector
                                tmp.push( a );
                    }

                    r = tmp;

                // We can get a speed boost by handling nth-child here
                }



                //来自API文档的描述:
                //匹配其父元素下的第N个子或奇偶元素
                //':eq(index)'
只匹配一个元素，而这个将为每一个父元素匹配子元素。:nth-child从1开始的
，而:eq()是从0算起的!
                //可以使用:
                //nth-child(even)
                //:nth-child(odd)
                //:nth-child(3n)
                //:nth-child(2)
                //:nth-child(3n+1)
                //:nth-child(3n+2)
                else if ( m[1] == ":" && m[2] == "nth-child" ) {
                    var merge = {}, tmp = [],
                        // parse equations like 'even', 'odd', '5',
'2n', '3n+2', '4n-1', '-n+6'
                        test = /(-?)(\d*)n((?:\+|-)?\d*)/.exec(

                            m[3] == "even" && "2n" ||           //" &&
"在JS中操作符的说明:依次获取每一个操作数，将它们转换为布尔变
                            m[3] == "odd" && "2n+1" ||
//量，如果是false，则直接返回这个操作数的值
（注意，返回的是转换前的原值，不一定
                            !/\D/.test(m[3]) && "0n+" + m[3] ||
//是布尔类型），中断后面操作数的处理；否则继续处理下一个操作数。如果直
到最后一个操作数仍
                            m[3]),
//然对应布尔变量true，则返回最后这个操作数的值

                        // calculate the numbers (first)n+(last)
including if they are negative
                        // 如果m[3]=="-2n+1",则test[1]=="-" ;
test[2]=="2" ; test[3]=="+1"
                        // 下面这个运算将上边分析出来的字符串转化成了数字
                        first = (test[1] + (test[2] || 1)) - 0, last =
test[3] - 0;

```

```
3357
                //获取完必要的参数之后,下面就要根据这些参数来查找(过滤)所需的元素了
3358
3359                // loop through all the elements left in the jQuery object
3360                //逐个逐个看看传进函数里面来的元素,看看是否符合需求
3361                for ( var i = 0, rl = r.length; i < rl; i++ ) {
3362                    var node = r[i], parentNode = node.parentNode, id = jQuery.data(parentNode);
3363                    //merge用来记录已经处理过的元素.
3364
                //如果id所指示的parentNode并没有被处理过,那就对这个parent的每一个孩子进行"普查",给他们每人一个"家庭编号"
3365                    //为什么要"普查"parent的孩子呢?
3366
                //因为nth-child的操作的作用就是"匹配其父元素下的第N个子或奇偶元素"
3367                    //其实 "普查" 的目的并不是所有的孩子,而是想知道node在这个parent的孩子中排第几
3368                    if ( !merge[id] ) {
3369                        var c = 1;
3370
3371                        for ( var n = parentNode.firstChild; n; n = n.nextSibling )
3372                            if ( n.nodeType == 1 )
3373                                n.nodeIndex = c++;
3374
3375                        merge[id] = true;
                //这个parent的孩子已经"普查"过了,记录下来
3376                    }
3377
3378                    var add = false;
3379
3380
                //"普查"完毕之后,这个parent的每个孩子都拿到了一个编号,下面就要根据这个编号,按照条件,决定是否把一个孩子加入结果集里面
3381
3382                    //first==0, 说明要的就是第last个孩子
3383                    if ( first == 0 ) {
3384                        if ( node.nodeIndex == last )
3385                            add = true;
3386                    }
3387
3388
                //下面是另一种情况.你的数学一定比我好,所有我就不多说了.
3389
                    //这个是为了保证nodeIndex跟last不相等
3390                    else if ( (node.nodeIndex - last) % first == 0 && (node.nodeIndex - last) / first >= 0 )
3391                        add = true;
3392
3393                    if ( add ^ not )//add跟not进行异或. not起一个开关的作用,为ture时,要的元素不要,不要的元素反而要...
3394                        tmp.push( node );
3395                }//end for
3396
3397                r = tmp;
3398
3399
```

```
3400                }
3401
3402                // Otherwise, find the expression to execute
3403                //t都不是要匹配的三种基本情况,那就放到这个else里面处理
3404                else {
3405
3406
//expr是一个集合,集合里面有好多的函数,使用m[1]设定的值来索引这些函数
3407                    var fn = jQuery.expr[ m[1] ];
3408
3409
//如果m[1]==":",那么fn还是一个对象,这个对象里面才是一堆的简单函数,使用m[2]再次索引出这些函数
3410                    if ( typeof fn == "object" )
3411                        fn = fn[ m[2] ];
3412
//如果函数是以字符串的形式给出,那么使用eval函数来使用它
3413                    if ( typeof fn == "string" )
3414                        fn = eval("false||function(a,i){return " + fn + ";}");
3415
3416                    // Execute it against the current filter
3417                    // 使用一个匿名函数来过滤元素.
3418                    // 可以看到匿名函数里面有使用了fn来做真正的过滤.
3419                    // 如果fn返回true,则把元素留下
3420                    // 如果fn返回false,则剔除元素.
3421                    //
注意,这些'ture留下false去掉'的过滤逻辑并不是一成不变的,当jQuery.grep函数的第三个参数为true
3422                    // 时,过滤逻辑就是'ture去掉false留下'了
3423                    r = jQuery.grep( r, function(elem, i){
3424                        return fn(elem, i, m, r);
//elem就是当前处理的匹配元素集合中的元素,i是它的索引.m是一个
3425
//数组,它装着对传入的选择器(t)进行正则匹配后的结果.r是经grep
3426
//处理过的当前结果集.只有jQuery.expr[':'].last函数使用了这个参数
3427
3428                    }, not );
//not如果为ture则翻转匿名函数的过滤逻辑:过滤函数原来那些要的元素就不要,不要的元素就要...
3429                }
3430
3431                //程序到了这里,
如果t里还有表达式,那就在进行一次循环.如此反复...
3432
3433            }//end while
3434
3435            // Return an array of filtered elements (r)
3436            // and the modified expression string (t)
3437            // 原文翻译: 将过滤得到的结果集(r)和修改过的表达式(t)返回
3438            return { r: r, t: t };
3439        },
3440
3441        /**
3442         * 以elem为起点,沿着dir所指定的路线返回指定的元素.
3443         * 有点晦涩,可能不太好理解,我举个例子:
3444         *
```

```
如elem是一个普通的DOM元素,而dir是'parentNode',那么dir函数就会返回[elem.parentNode,elem.parentNode.parentNode...].
3445        *
又比如,elem是一个普通的DOM元素,dir则是'nextSibling',那么dir就会返回[elem.nextSibling,elem.nextSibling.nextSibling...].
3446        *
注意,这种一层一层的返回不是没有限制的,当要处理的元素为document时,操作停止.
3447        *
3448        * @param {HTMLElement} elem
一个普通的DOM元素.实际上它的类型并不限于HTMLElement,XML的DOM对象也可以.
3449        * @param {string} dir
字符串,'parentNode','nextSibling','previousSibling'三者其一.
3450        */
3451       dir: function( elem, dir ){
3452           var matched = [],//结果集
3453               cur = elem[dir];//初始化cur为elem[dir]
3454           while ( cur && cur != document ) {
3455               if ( cur.nodeType == 1/* Node.ELEMENT_NODE */ )
3456                   matched.push( cur );
3457               cur = cur[dir];//cur指向下一个目标.
3458           }
3459           return matched;
3460       },
3461
3462       /**
3463        *
3464        *
以cur指定的元素为起点,一直调用'cur[dir];cur=cur[dir]',直到调用的次数等于result并且cur是节点元素为止.下面这行代码是一个经典调用场景:
3465        * return jQuery.nth(a.parentNode.lastChild,1,"previousSibling")==a;
3466        *
3467        * 如果还是不知道我在说什么,建议详细查看代码.
3468        *
3469        * @param {HTMLElement} cur
一个普通的DOM元素.注意,不限于HTMLElement元素,XML的DOM元素可以.
3470        * @param {number} result
一个数字.当处理到第result个元素的时候,函数停止并返回结果.
3471        * @param {string} dir
一个字符串,为'previousSibling'和'nextSibling'两者其一.
3472        * @param {Object} elem 可以看到在本函数中并没有用到这个参数.
3473        */
3474       nth: function(cur,result,dir,elem){
3475           result = result || 1;//知道也要处理一个
3476           var num = 0;
3477
3478           for ( ; cur; cur = cur[dir] )
3479               if ( cur.nodeType == 1/* Node.ELEMENT_NODE */ && ++num == result )
3480                   break;
3481
3482           return cur;
3483       },
3484       /**
3485        * 选择elem的所有兄弟节点.
3486        * 这个函数的代码已经很精练,不太好详细解释.请自行品味.
```

```
3487        *
3488        * @param {HTMLElement} n
它给elem查找自己的兄弟划定了一个范围.elem的兄弟必须在n里面找.注意,其实并不限于HTML文档,XML内的元素也可以使用本函数
3489        * @param {HTMLElement} elem
除了这个元素之外,所有n中的兄弟节点都会被加进结果集.注意,也不限于HTML元素,XML元素亦可.
3490        */
3491       sibling: function( n, elem ) {
3492           var r = [];
3493           for ( ; n; n = n.nextSibling ) {
3494               if ( n.nodeType == 1 /* Node.ELEMENT_NODE */ && n != elem )
3495                   r.push( n );
3496           }
3497           return r;
3498       }
3499   });//extend 结束
3500
3501
3502
3503
3504
3505
3506
3507
3508
3509
3510   //-----------------------------------------------------------------
下面对jQuery进行事件方面的扩展 ------------------------------
3511   /*
3512    * A number of helper functions used for managing events.
3513    * Many of the ideas behind this code orignated from
3514    * Dean Edwards' addEvent library."
3515    */
3516   /*
这里是一系列的事件方法.正如上面那段英文所说的那样,这些代码的许多思想并不是John
Resig自己原创的.从这我们可以到,站在巨人的肩膀之上效益是非常大的.
3517    * 好了,说正事:
3518    *
这里定义的方法都是定义在jQuery.event命名空间上的静态方法.他们并直接不向jQuery库用户公开(当然你有权强行调用他们,只要对他们足够了解).这些方
3519    * 法主要是被jQuery对象上的与事件相关的方法调用.
3520    *
3521    * 注意, 下面的中文注释中 "事件监听函数" 与 "事件处理函数"
的语义是一致的, 尽管它们是不同的东西...请读者注意.
3522    */
3523   //-----------------------------------------------------------------
-----------------------------------------------
3524
3525   jQuery.event = {
3526
3527       // Bind an event to an element
3528       // Original by Dean Edwards
3529       /**
3530        * 为元素添加一个事件监听器.
3531        *
```

```
3532        * @param {HTMLElement} elem 元素, 就是要为它添加一个事件监听器
3533        * @param {String} types
    字符串,表示要注册的事件类型,如'click','moveover'等.
3534        * @param {Function} handler
    callback函数,事件发生后,此函数将会被调用.
3535        * @param {Object} data
3536        */
3537       add: function(elem, types, handler, data) {
3538
3539           // 8 是comment类型的节点,  3 是 text
    节点,这些节点就不需要添加什么事件监听器了.
3540           if ( elem.nodeType == 3/* Node.TEXT_NODE */ || elem.nodeType
    == 8 /* Node.COMMENT_NODE */ )
3541               return;
3542
3543           // For whatever reason, IE has trouble passing the window
    object
3544           // around, causing it to be cloned in the process
3545           // 不知道咋嘀, 在IE浏览器中,
    如果把window对象作为函数参数传递的话往往不能正确传递(即函数内部根本不
    知道它是window对象).于是利用下面这
3546           //
    个if来判断是不是在IE浏览器中以及检查传进来的elem会不会可能是window对象
    , 如果是, 就让elem引用重新指向window.
3547           if ( jQuery.browser.msie && elem.setInterval )
    //elem.setIntervale是想通过这个检测elem是不是window对象。
3548               elem = window;
3549
3550           // Make sure that the function being executed has a unique ID
3551           // 翻译: 确保每一个事件处理函数(也就是handler)都有一个ID.
3552           if ( !handler.guid )
3553               handler.guid = this.guid++;
    //this指向的是jQuery.event这个命名空间,它本质就是一个对象.guid的初始值
    为1.
3554
3555           // if data is passed, bind to handler 翻译:如果传入了data,
    那么就把这些data绑定到handler上
3556           if( data != undefined ) {
3557               // Create temporary function pointer to original handler
3558               // 把handler的引用传给一个临时的变量fn.
    等下要在它的外面再套一层.
3559               var fn = handler;
3560
3561               // Create unique handler function, wrapped around
    original handler
3562               //
    翻译:创建一个唯一的处理函数,这个函数包裹着原来的那个处理函数.
3563               // this指向的仍然是jQuery.event对象.
    this.proxy函数最后返回的仍然是它的第二个参数中的那个新创建的匿名函数.
    this.proxy的作用是
3564               //
    让匿名函数具有fn一样的ID,仅此而已(可以参考jQuery.event.proxy函数的中文
    注释.). 隐藏在proxy背后的重要思想是"包裹"函数
3565               //(wrapper function), 具体请留意下面的中文注释.
3566               handler = this.proxy( fn, function() {
3567                   // Pass arguments and context to original handler
3568                   // 翻译: 传递参数和上下文给原来的那个函数(即fn)
3569
```

```
3570                    /*
3571                     *
        这种将一个函数使用另外一个函数"包裹"起来的技术在jQuery中十分常见,
        它的目的就是通过闭包机制来传递参数以及上下文. 以此来达到改变
3572                     * 传递的参数或者改变调用上下文的目的.
3573                     * 具体来讲一下:
3574                     * 当fn最初被作为一个函数的引用传进来的时候,
        它所作用的上下文不一定是jQuery.event. 比如说当我们在全局的作用域中
3575                     * 定义了一个事件处理函数,
        这个函数中的代码中如果含有this关键子,
        那么这个this就可以引用window对象, 因为这个全局的函数最终将会
3576                     * 作为window对象的方法来调用.
        所以如果我们想要将fn作为其他对象的方法来调用,
        就是必须使用apply(或者call)方法来改变函数的执行上下文.
3577                     * 而这里就是想改变fn的执行上下文,
        想将处理函数作为触发事件的那个元素的方法来调用.
        注意以下代码的this关键字的含义.
3578                     */
3579
3580                    // 实际上还是调用原来那个处理函数来处理事件,
        不过使用了apply函数传递了this作为新的函数执行上下文.注意下面的this指向
3581                    // 一个HTMLElement元素, 即经过这么一层"包裹"之后,
        fn改变了自己的执行上下文为这个HTMLElement,
        它内部代码中this全部指向的是
3582                    // 这个HTMLElement元素.
        其实要想确定这个this到底指向的是什么,
        需要追溯到最终函数是怎样被调用的.具体可以查看
3583                    // jQuery.event.trigger函数的中文注释部分.
3584                    return fn.apply(this, arguments);
3585                });
3586
3587                // Store data in unique handler
3588                // 将数据存储到handler上面. handler从创建到现在,
        具有了与fn相同的函数功能, 与fn一样的guid, 以及用户传进来的data.
        而不同的就是调用的上下文
3589                handler.data = data;
3590            }
3591
3592            // Init the element's event structure
3593            // 翻译: 初始化 element的 event数据结构.
3594            /* 继续阅读以下代码, 必须首先了解以下jQuery的事件机制:
3595             * 首先, jQuery的事件分为原生事件与jQuery自定义事件.
        像click, blur, focus等事件,
        都是原生事件.这些事件的触发由系统自动完成; 而自定义
3596             * 事件的触发则需要手动来完成. 如jQuery的ajaxStart,
        ajaxStop等事件都是jQuery自己定义事件, 因为JavaScript并不提供这些事件.
3597             *
3598             * jQuery为了统一这两种事件, 推出了自己的add(注册) ->
        trigger -> handdle -> remove 事件模型.(虽然这些ideas来自Dean Edwards)
3599             *
3600             * 在JavaScript中,
        事件与监听函数的管理由JavaScript本身来完成,
        像"我的监听函数保存在哪?"这样的问题, 你根本不需要理会. 而jQuery为了获
3601             * 取更灵活的事件扩展,
        它在JavaScript事件机制基础之上扩展了自己的事件机制.
        在这个事件机制中,jQuery自己管理那些注册在某个事件上等待触发的监听函数.
3602             * jQuery事件机制具体内容是这样的:
3603             * (1)  注册: 使用jQuery.event.add函数完成.
```

```
add会将监听函数保存到元素的events缓存区中.
如果事件是首次注册监听函数, 那么jQuery就会再
3604            *       为元素开辟多一块叫handle的数据缓存区.
这个区仅仅装一个类型为jQuery.event.handle的函数. 这个函数是一个代理,
elem上所有事件被
3605            *       触发后都首先调用这个函数,
进行一些事件模型的兼容性处理(还有其他)之后,
这个handle函数就会从元素的events缓存区中选择应该运行的监听
3606            *       函数来运行.
3607            * (2)  触发:
jQuery自定义事件需要jQuery自己在合适的时机手动触发. 如ajaxStop事件,
jQuery检查当前ajax请求数, 当检查到这个请求数为0时,
3608            *
jQuery就调用jQuery.event.trigger自己手动触发ajaxStop事件.
3609            *
而对于原生事件,则通过将代理handle函数注册为原生的事件监听器来触发.
也举一个例子: 比如click事件. jQuery使用
3610            *
addEventListener/attachEvent将代理的handle函数注册到了JavaScript的原生
事件机制中. 当click事件发生时, 代理handle函数
3611            *       就会被调用.
3612            * (3)  处理: jQuer.event.handle
会在elem的所有事件监听器被调用之前运行.它的工作主要是作为一个代理,
屏蔽事件模型的浏览器兼容性问题,
3613            *       检查事件的命名空间,
最后从元素的evnets监听函数列表中, 逐个执行需要触发的监听函数.
3614            * (4)  卸载:
原生事件需要removeEventListener/detachEvent来移除注册在事件的上的代理h
andle函数. 上面说过, 两种事件类型的事件监听函数
3615            *       被jQuery存储在elem的events数据缓存区中,
于是卸载事件监听函数时,
jQuery就会从elem的events数据缓存区中删除该函数的引用. 如果
3616            *       jQuery发现events为空了,
说明elem已经没有事件监听函数,
events数据缓存区和handle数据缓存区将被移除.
3617            *
3618            *
3619            */
3620
3621
3622          // 准备取得elem的缓存数据( jQuery 可以让每一个元素"缓存"数据
)",这些数据用"events"做为键值来获取。如果" events " 并不对应有数据
3623          // 就让 " events " 初始化为
{}.注意jQuery.data的用法:当没有传入第三个参数时为获取元素的缓存数据.若
传入了第三个参数,那么则是设置元素的
3624          // 缓存数据.
events可以使用各种事件类型的名称作为键值来存取内容, 如click,
mouseover等, events['click']获得的是处理click事件处理函
3625          // 列表.
3626          var events = jQuery.data(elem, "events") || jQuery.data(elem,
 "events", {}),
3627
3628                                                  //如果"||"
左边的代码不能获取elem的handle,那就说明目前还没handle,那就自己初始化一
个
3629              handle = jQuery.data(elem, "handle") || jQuery.data(elem,
 "handle", function(){
3630                  // Handle the second event of a trigger and when
```

```
3631                // an event is called after a page has unloaded
3632                if ( typeof jQuery != "undefined" && !jQuery.event.
    triggered )
3633                    return jQuery.event.handle.apply(arguments.callee
    .elem, arguments);
3634            });
3635
3636
3637        // Add elem as a property of the handle function
3638        // This is to prevent a memory leak with non-native
3639        // event in IE.
3640        handle.elem = elem;
3641
3642        // Handle multiple events separated by a space
3643        // jQuery(...).bind("mouseover mouseout", fn);
3644        /*
3645         * 处理多事件的情形,这些事件使用" "号隔开.
3646         * 调用jQuery函数的静态each函数,
    对每一个事件类型使用一个函数进行处理.至于处理的内容是什么,
    那就请往下看了
3647         */
3648        jQuery.each(types.split(/\s+/), function(index, type) {
3649            // Namespaced event handlers
3650            var parts = type.split(".");
3651
3652            /* 这里要对上面的"Namespaced event handlers"进行说明:
3653             * "Namespaced
    event"是jQuery"原创"的概念,旨在将某种特定的事件类型的监听函数进行更加
    细的划分,从而达到更加灵活地删除,触发事件监听函
3654             * 数的目的.
    而jQuery通过为事件类型添加".namespace"的方式来对一个事件类型的不同监听
    函数进行划分. 如"click.myClick",
3655             *
    "click.yourClick",这样我们当我们需要卸载或者激活某类特定的Click事件监
    听器时可以这么写:$('.someClass').unbind('click.yourClick').
3656             *
    这样使用"click.yourClick"绑定的事件处理函数就会被移除,但是使用"click.m
    yClick"注册的事件处理函数就被保留.
3657             *
3658             * 更多有关"Namespaced event
    handlers"的信息,请参考jQuery的官方网站, 以下是链接:
3659             * http://docs.jquery.com/Namespaced_Events
3660             */
3661
3662            type = parts[0];
    //part[0]是"."左边的部分,如"click.yours"中的"click"
3663            handler.type = parts[1];
    //parts[1]是事件的命名空间,如"click.yours"中的"yours"
3664
3665            // Get the current list of functions bound to this event
3666            // 翻译:获取当前绑定在这个事件上的处理函数列表.
3667            var handlers = events[type];
3668
3669            // Init the event handler queue
3670            // 如果并不存在type所指示的事件类型的处理函数列表,
    就自己创建一个.
3671            if (!handlers) {
3672                handlers = events[type] = {};
```

```
3673
3674                    // Check for a special event handler
3675                    // Only use addEventListener/attachEvent if the
    special
3676                    // events handler returns false
3677                    /* 先翻译:检测特殊事件(ready, mouseenter,
    mouseleave)的处理函数,如果特殊事件的处理函数返回false那就只用
3678                     * addEventListener/attachEvent 来添加处理函数
3679                     *
3680                     * 好,现在进行说明:
3681                     * 所谓的特殊事件也就三种:ready,mouseenter,mouseleave
3682                     *
    ready事件并不是浏览器所支持的事件(即浏览器中没有类似"onReady"这样的事
    件),ready是DOM Ready之意,有些浏览器有"DOMContentLoaded"事件与之相对应.
3683                     *
    由于DOMContentLoaded事件并不普及,因此不能通过统一的addEventListener/at
    tachEvent来添加这个事件.jQuery中使用bindReady来将你的函数绑定到
3684                     * DOMContentLoaded事件当中,
    因此不需要使用addEventListener/attachEvent来绑定事件,也就是说不用下面
    的代码来做这件事情. 下面的代码(整个if语句)是
3685                     *
    用来为一般的事件(如mouseover,click等)添加处理函数的.
    由于IE和w3c采用了不同的事件模型,因此if内的语句又有if对两种事件模型进行
    了区分.
3686                     *
3687                     *
    mouseenter和mouseleave是IE所支持的事件,w3c所支持的对应事件是mousein和m
    ouseout.如果特殊事件是mouseenter和mouseleave中的一个,并且当前浏览器
3688                     *
    就是IE,那么也由下面的if语句体来完成事件处理函数的绑定.
    如果当前浏览器不是IE但是又要求绑定这两个事件,
    那么"jQuery.event.special[type].setup.call(elem)"
3689                     * 的返回值不会是false,
    那么程序也就是不会用下面的代码进行事件的绑定, 而是在
    jQuery.event.special[type].setup 函数内部使用jQuery对象的
3690                     * bind函数, 绑定与 mouseenter和mouseleave
    相对应的w3c事件,即mouverover 和mouseout.
3691                     *
3692                     * 另外可以参考jQuery.event.special中的中文注释.
3693                     */
3694                    if ( !jQuery.event.special[type] || jQuery.event.
    special[type].setup.call(elem) === false ) {
3695
                            //将setup作为elem的方法调用, 于是setup函数中的
3696
                            //this关键字就会被替换成elem的引用.
3697
3698
3699                        // Bind the global event handler to the element
3700                        if (elem.addEventListener)// FF
3701                            elem.addEventListener(type, handle, false);
3702                        else if (elem.attachEvent)// IE
3703                            elem.attachEvent("on" + type, handle);
3704                    }
3705                }
3706
3707                // Add the function to the element's handler list
3708                // 元素 elem 对每一种类型的 event 都有若干的 handler. 而
```

```
handlers 就是这些 handler 的集合. 每一个handler使用它自己的guid来索引.
3709            handlers[handler.guid] = handler;
3710
3711            // Keep track of which events have been used, for global
triggering
3712            // 记录下到底哪些事件被使用( 或者说被监听 )了,
在trigger函数中需要用到这个属性
3713            jQuery.event.global[type] = true;
3714        });
3715
3716        // Nullify elem to prevent memory leaks in IE
3717        // 将elem的引用设置为null,避免在IE中导致内存泄漏.
3718        elem = null;
3719    },
3720
3721    guid: 1,
3722    global: {},
3723
3724    // Detach an event or set of events from an element
3725    /**
3726     * 卸载一个事件,或者一个事件集合.
3727     *
3728     * @param {HTMLDOMElement} elem
3729     * @param {Object} types - 可能是字符串, 也可能是一个事件对象.
3730     * @param {Function} handler
3731     */
3732    remove: function(elem, types, handler) {
3733        // don't do events on text and comment nodes
3734        if ( elem.nodeType == 3/* Node.TEXT_NODE */ || elem.nodeType
== 8/* Node.COMMENT_NODE */ )
3735            return;//如果是文本节点和注释节点,就不用干活了,直接返回.
3736
3737        var events = jQuery.data(elem, "events"),
//获取为elem所缓存的索引为"events"的数据.
事实上这些数据就是为elem所注册的事件监听函数
3738            ret, index;
3739
3740        //
如果在elem元素上面是有events的, 也即元素有对某些事件进行监听的,就可以
继续进行事件监听函数的卸载工作;如果events值为空,则函数返回.
3741        if ( events ) {
3742            // Unbind all events for the element
3743            // 如果types为undefined,
或者说types这个字符串以"."开头,就卸载元素上的所有事件监听函数.
3744            if ( types == undefined || (typeof types == "string" &&
types.charAt(0) == ".") )
3745                for ( var type in events )
3746                    this.remove( elem, type + (types || "") );
3747
3748            // 如果传入的不是字符串, 并且不为 undefined (
不是用字符串来表示要 remove 的操作 )
3749            else {
3750                // types is actually an event object here
3751                // 如果传入的 types 就是一个事件对象( 就是 event ||
window.event 这个对象 ), 那么就把 handler 和 types的引用 " 矫正 " 过来
3752                // 这样下面的代码就不用修改,还是照样能用了
3753                if ( types.type ) {
//通过检查types上有没有type属性来判断types是不是一个event对象
```

```
3754                        handler = types.handler;
//types上的handler属性是在jQuery.event.handle函数中添加上的,目的就是为了方便我们在这里把它删
3755
//除.详细参见jQuery.event.handle函数
3756                        types = types.type;
//让types真真正正指向一个事件类型.
3757                    }
3758
3759                    // Handle multiple events seperated by a space
3760                    // jQuery(...).unbind("mouseover mouseout", fn);
3761                    /*
3762                     * 翻译:
处理用空格隔开的卸载多个事件的事件监听器材的情况.如
jQuery(...).unbind("mouseover mouseout", fn);
3763                     */
3764                    jQuery.each(types.split(/\s+/), function(index, type){
3765                        // Namespaced event handlers
3766
3767                        /*
3768                         * 如果你对" Namespaced event handlers "不了解,请到jQuery.event.add中查看相应的中文注释.
3769                         */
3770
3771                        var parts = type.split(".");
3772                        type = parts[0];//part[0]是事件类型
3773
3774                        // 还记得前面jQuery.event.add函数中的 handlers 吗? events[ type ] 就等于 handlers
3775                        if ( events[type] ) {
//如果在type所指定的事件上有绑定事件监听函数,
那就视情况而定删掉对应的监听函数.
3776                            // remove the given handler for the given type
3777                            //
如果传入的handler是有引用的，表示仅仅需要删除绑定在type所指定事件上的handler所指向那个监听函数.那就只除去这个handler咯
3778                            if ( handler )
3779                                delete events[type][handler.guid];
//使用函数的唯一id将函数删除
3780
3781                            // remove all handlers for the given type
3782                            // 如果没有传入handler,
而仅仅是传入了elem,types，说明要除去对应事件上所有 handler
3783                            else
3784                                for ( handler in events[type] )
3785                                    // Handle the removal of namespaced events
3786                                    /* 删除的时候要注意啦:
3787                                     *
(1)如果parts[1]是空值,说明我们并没有使用命名空间对某种事件类型的监听函数进行进一步的划分.那么我也就可以放心地
3788                                     * 删除type所指定事件下的所有监听函数.
3789                                     *
(2)如果parts[1]是有值的,那么就仅仅删除那些命名空间与parts[1]相同的handler. 注意,下列代码中的
3790                                     * "events[type][handler].type "内的"type"属性是在jQuery.event.add时加上的.
3791                                     */
```

```
3792                                    if ( !parts[1] || events[type][
handler].type == parts[1] )
3793                                        delete events[type][handler];
3794
3795                                // remove generic event handler if no more
handlers exist
3796                                /*
以下这个for循环不断地从events[type]取属性名出来,并放入到ret中.
3797                                 *
一旦name获得了一个可以转化为true的值,for的循环体就会被执行.执行之后
3798                                 *
居然是break...于是就起到了一个作用:检测元素是否还有自定义的属性.
3799                                 * 注意,元素的继承属性不能被for
in循环枚举.例如从Object中继承下来的
3800                                 * toString函数就不能被for in访问到.
3801                                 */
3802                                for ( ret in events[type] ) break;
3803                                if ( !ret ) {
//!ret为true时说明events[type]上已经没有了事件监听器了.
3804
3805                                    // 如果 type 表示的并不是特殊事件( 如
ready ),
或者是特殊事件,但teardown函数认为这个特殊事件可以使用下面的简便
3806                                    // 方法来卸载事件处理函数,
而没有必要让它来干这事情,
那就是使用下面的代码(if语句体内代码)来直接卸载.如果有兴趣想知道
3807                                    //
为什么teardown函数会认为"没必要让我来卸载这个事件上的监听函数",请参考j
Query.event.special内的teardown函数的中文注释
3808                                    if ( !jQuery.event.special[type] ||
jQuery.event.special[type].teardown.call(elem) === false ) {
3809                                        if (elem.removeEventListener)// FF
3810                                            elem.removeEventListener(type,
jQuery.data(elem, "handle"), false);
3811                                        else if (elem.detachEvent) // IE
3812                                            elem.detachEvent("on" + type,
jQuery.data(elem, "handle"));
3813                                    }
3814
3815
3816                                    ret = null;
3817                                    // 整个 type
代表的事件下的所有事件handler都不要了, 因为前面 !ret
成立表示该集合为空了
3818                                    delete events[type];
3819                                }
3820                            }
3821                        });
3822                }
3823
3824                // Remove the expando if it's no longer used
3825                // 翻译:
如果events不再被使用,那就删除那些expando(非继承的)属性.
3826
3827
3828                for ( ret in events ) break;
3829                if ( !ret ) {
//如果ret仍然为undefined(定义变量的初始值),说明events是空的. events
```

```
空了, 表示这个元素再也没有任何的监听事件.
3830                var handle = jQuery.data( elem, "handle" );
3831                if ( handle ) handle.elem = null;
3832                jQuery.removeData( elem, "events" );//
移除elem上的事件监听函数列表, 因为它是空的.
3833                jQuery.removeData( elem, "handle" );//
移除代理的事件处理函数.
3834            }
3835        }
3836    },
3837    /**
3838     * trigger函数要分三步做三件事情:
3839     * (1) 触发通过jQuery.event.add注册的监听函数
3840     * (2) 执行用户传入的extra函数
3841     * (3) 触发本地的事件处理函数(即直接写在HTML标签内的函数)
3842     *
3843     * trigger函数最后返回一个布尔值,
浏览器可以根据这个布尔值来决定是否进行默认行为.
而这个值到底是什么(true or false), 由上面所列的三步处理共
3844     * 同决定. 此外, trigger函数返回undefined也是允许的.
3845     *
3846     * @param {string} type - 事件类型
3847     * @param {Array} data - 需要传给事件监听函数的数据
3848     * @param {HTMLElement} elem - 发生事件元素
3849     * @param {boolean} donative - donative 其实为"do native",
是否执行本地方法(即直接写在HTML标签中的事件处理函数)
3850     * @param {Function} extra - 用户需要在触发事件处理函数之后,
需要再运行的函数. 这个函数的执行能够影响trigger最终返回布尔值.
3851     */
3852    trigger: function(type, data, elem, donative, extra) {
3853        // Clone the incoming data, if any
3854        data = jQuery.makeArray(data);
//makeArray函数将data转换为一个真正的数组.
3855
3856        // 传入的事件类型 type 竟然会饱含有字符" ! " !?
其实这表示的是一个 ! ( not ) 的操作. 如 !click 就是除了 click
之外的事件
3857        if ( type.indexOf("!") >= 0 ) {
3858            type = type.slice(0, -1); // 相当于type = type.slice( 0,
length+(-1) ); 去掉最后一个字符
3859            var exclusive = true; // 设这个变量为true
告诉程序type中含有 " ! "
3860        }
3861
3862        // Handle a global trigger 翻译:处理有一个全局的触发器
3863        // 如果elem是空的,
那么就认为是要给触发所有元素上的绑定在type所指定事件上的所有监听函数.
3864        if ( !elem ) {
3865            // Only trigger if we've ever bound an event for it
3866            // 在 add 函数中最后不是有一句 jQuery.event.global[type]
= true 吗? 这时候派上用场了
3867            // 如果if内的语句为true, 这说明 type
表示的事件已经被监听,需要为这个事件加入触发器( trigger )
3868            if ( this.global[type] )
3869                                //
在jQuery对象的类数组中加入window,document对象
3870                jQuery("*").add([window, document]).trigger(type,
data);
```

```
3871                                                    //
    然后为这些对象添加触发器( trigger )
3872
3873
3874            }
3875            // Handle triggering a single element
3876            // 如果不是空的,
    那就是要触发单个元素的绑定在type所指定的事件上的事件监听函数.
3877            else {
3878                // don't do events on text and comment nodes
3879                if ( elem.nodeType == 3 /* Node.TEXT_NODE */|| elem.
    nodeType == 8 /* Node.COMMENT_NODE */ )// 3 是text node, 8 是comment
    node
3880                    return undefined;
3881
3882                var val,
3883                    ret,
3884                    fn = jQuery.isFunction( elem[ type ] || null ),
3885                    // Check to see if we need to provide a fake event,
    or not
3886
3887                    /* 由于最后的data需要传给事件监听函数,
    而标准事件模型中要求事件监听函数的第一个参数必须是event对象,即data[0]
    必须是event对象,
3888                     * 于是我们必须检测data的[0]是不是event对象. 是,
    那当然好; 如果不是, 我们则在data的头部加入一个伪造的(fake)event对象.
3889                     */
3890                    event = !data[0] || !data[0].preventDefault;//
    preventDefalt是 w3c的标准方法,它是event对象的方法.
3891                                                            //
    如果不能检测data[0]有这个方法则证明data[0]不是event对象.
3892                // Pass along a fake event
3893                // OK, 如果不幸发生了(data[0]不是event对象),
    那我们自己给它加上一个.
3894                if ( event ) {
    //data[0]并不是w3c标准的事件对象,那就新建立一个对象,把一个event对象该
    有的一些方法和属性都加入到这个对象中,并用
3895                                //unshift方法把对象加在data数组的头部.
3896                    data.unshift({
3897                        type: type,
3898                        target: elem,
3899                        preventDefault: function(){},
3900                        stopPropagation: function(){},
3901                        timeStamp: now()
3902                    });
3903
3904                    //expando 只是一个由 " jQuery " + now()
    组成的字符串, 以此字符串作为一个属性名,
    是为了说明这是一个jQuery处理过的事件对象
3905                    data[0][expando] = true; // no need to fix fake
    event 好了,事件对象已经处理过了,它是我们自己加上去的, 作个标记. 以后
3906                                                    //
    data可能会被传入到jQuery.event.fix函数中,
    fix函数看到expando这个属性, 它就会略过一些工序.
3907                                                    //
    详细信息请参考jQuery.event.fix函数的中文注释.
3908                }
3909
```

```
3910                // Enforce the right trigger type
3911                // 将事件类型强制修改为正确的类型.
3912                data[0].type = type;
3913
3914                // 上面设置的标志: 是否含有" ! " 字符
3915                if ( exclusive )
3916                    data[0].exclusive = true;//在事件对象上加入属性exclusive.
3917
3918                // Trigger the event, it is assumed that "handle" is a function
3919                // 翻译: 触发事件, "handle"被假设是一个函数
3920                // 好, 第一步:运行通过jQuery.event.add注册的函数.
3921                var handle = jQuery.data(elem, "handle");//通过elem的数据缓存区, 获取elem的监听函数.
3922                if ( handle )//如果在这个缓存区上有监听函数,那么就将这个监听函数作为elem的方法来运行.
3923                                /* 注意,elem的数据缓存区中缓存的这个handle并不是它传给jQuery.event.add注册的那个handle.
3924                                 * 这个handle的类型是jQuery.event.handle.它的作用就是运行elem绑定在某个事件上的所有事件监听函数.
3925                                 * 也就是说,当我们通过jQuery.event.add注册一个事件监听函数时,add函数内部将这个监听函数放进某个事件的监听函数列
3926                                 * 表当中.比如说jQuery.event.add('click',function(){//do some thing}),函数将被加入到elem的click事件
3927                                 * 监听函数列表, 即events['click'].而这个events列表被缓存在jQuery的数据缓存区中,可通过jQuery.data(elem,'events')
3928                                 * 获取. 然后, 如果这是elem首次添加监听函数,add函数就再给elem 开辟一个数据缓存区, 并在这个缓存区上只存一个函数. 这
3929                                 * 个函数就是现在我们在这里获取的这个handle.
3930                                 *
3931                                 * 任何一个事件被触发时,jQuery总是到元素的数据缓存区去获取这个handle,再由这个handle作为一个代理, 在解决事件模型的兼容性
3932                                 *问题后,逐一触发elem绑定在该事件上的事件监听函数.
3933                                 */
3934                    val = handle.apply( elem, data );//好了,执行代理handle函数, 绑定再elem上的事件处理函数将会被执行.
3935
3936                    /*
3937                     * 这里对上面的val再补充一些说明:
3938                     *由于jQuery.event.handle作为所有的监听函数的代理函数,因此除了要运行实际的事件监听函数之外, 还要代替原来的事件监听函数与浏览
3939                     * 器打交道.
3940                     *
3941                     * 在没有代理的时候,事件监听函数在函数执行完毕之后将返回一个布尔值来允许(true)或者取消(false)浏览器的默认行为. 那么当代理代替了
3942                     * 原来的监听函数之后,代理函数理所当然要返回同样的一个布尔值(val)来与浏览器产生互动.
3943                     *
3944                     * 下面的代码将围绕这个布尔值(val)展开工作.
```

```
3945                 */
3946
3947
3948
3949             // Handle triggering native .onfoo handlers (and on
    links since we don't call .click() for links)
3950             // 关于fn,上面有代码: fn = jQuery.isFunction( elem[ type
    ] || null )
3951             if ( (!fn || (jQuery.nodeName(elem, 'a') && type ==
    "click")) && elem["on"+type] && elem["on"+type].apply( elem, data )
    === false )
3952                 val = false;//如果没有本地函数, 那就返回false.
3953
3954             // Extra functions don't get the custom event object
3955             // 翻译: Extra函数并不需要用户事件对象
3956             // 由于Extra是用户自己提供的函数而不是一个事件监听函数,
    因此不需要事件对象,即不需要data[0].
3957             if ( event )
3958                 data.shift();// 移除第一个元素data[0]
3959
3960             // Handle triggering of extra function
3961             // 运行extra函数.
3962             if ( extra && jQuery.isFunction( extra ) ) {
    //如果用户传入了这个extra函数并且extra真的是一个函数.
3963                 // call the extra function and tack the current
    return value on the end for possible inspection
3964                 // 翻译:
    调用extra函数并且将当前的trigger返回值加入到extra函数的参数列表的后面
3965                 ret = extra.apply( elem, val == null ? data : data.
    concat( val ) );//运行这个extra函数
3966                 // if anything is returned, give it precedence and
    have it overwrite the previous value
3967                 // 翻译: 如果有任何返回值,
    给予优先级让它覆盖掉原来的val的值.
3968                 if (ret !== undefined)
3969                     val = ret;
3970             }
3971
3972
3973
3974             // Trigger the native events (except for clicks on links)
3975             // 翻译: 触发本地事件(除了link的click事件)
3976             /* 上面所说的 "native events"
    指的是直接写在HTML标签内的事件处理函数.
3977              * 在执行完jQuery所注册的事件处理函数之后,
    如果有,那就再继续执行 "native events" 的函数.
3978              * donative是一个开关, 它参与决定是否要执行触发本地事件.
3979              */
3980             if ( fn && donative !== false && val !== false && !(
    jQuery.nodeName(elem, 'a') && type == "click") ) {
3981                 //this在这里指的是jQuery.event
3982                 this.triggered = true;
3983                 try {
3984                     // 执行elem 上的 [type] 方法
3985                     elem[ type ]();
3986                 // prevent IE from throwing an error for some hidden
    elements
3987                 // 翻译:
```

```
                    防止IE在一些隐藏的元素上执行本地事件处理函数会抛出错误的情况.
3988                    } catch (e) {}
3989                }
3990
3991                this.triggered = false;
3992
3993            }//为单个元素触发监听函数结束
3994
3995
3996            //返回 val 它是一个布尔值.
                浏览器可以根据这个值来决定自己是否执行默认行为.
3997            return val;
3998        },
3999
4000        /**
4001         * 执行event所指定事件类型下的, 与触发元素有关的所有事件监听!
4002         * jQuery的事件机制要求触发的所有事件必须首先运行这个函数,
                由它做一些与兼容性, 命名空间相关的工作之后,
                再由它来代理运行绑定在指定事件上的所有
4003         * 应该运行的事件监听函数.
4004         *
4005         * 注意本函数的参数event对象, 它由jQuery传入,
                而不是由原生的JavaScript事件机制传入.
4006         *
4007         * @param {Event} event - 事件对象
4008         */
4009        handle: function(event) {
4010            // returned undefined or false
4011            var val, ret, namespace, all, handlers;
4012
4013            //
                IE和w3c在事件对象模型上有比较大的差异.首先是两者的事件对象所处的作用范围的不同,其次就是事件对象本身的属性也不尽相同.
4014            /* (1)
                作用范围的不同:在IE中,事件对象(event)是作为window的属性而存在的.即window.event可以获取到事件对象的引用.由于浏览器一次只处理
4015             *
                一个事件,因此这种设计本身虽然奇怪但并不会引起什么问题.而在w3c中,事件对象则在事件句柄被调用时作为第一个参数传入.如click(e)中的参数'e'
4016             * 实际上就是一个事件对象event的引用.
4017             * (2)
                在两种事件模型中,event对象的属性也不尽相同.如经典的事件源对象在IE中使用event.srcElement引用,而w3c则采用event.target.其他的
4018             * 就不一一列举.
4019             *
4020             *
                其实,w3c的事件模型是参考IE的事件模型制定出来的...唉,IE与w3c标准之间的'恩怨'真的是说不清理还乱...
4021             *
4022             *
                基于以上事实,为了让代码能够有一个统一的行为,以下这行代码就调用jQuery.event.fix来给event对象做个'修理'.具体fix里面做了什么修理,可以
4023             * 参考jQuery.event.fix函数的中文注释部分.
4024             */
4025            event = arguments[0] = jQuery.event.fix( event || window.
                event );
4026
4027            // Namespaced event handlers //关于"Namespaced event
```

```
handlers" 请Ctrl + F, 在其他函数的中文注释中有说明.
4028            namespace = event.type.split(".");
4029            event.type = namespace[0];
//namespace[0]就是真正的type,它的值可能是'click','load'等.
4030            namespace = namespace[1];//在namespace[0]后面的'命名空间'
4031
4032            // Cache this now, all = true means, any handler
4033            //
如果没有命名空间并且这个事件并没有被设置为'排除(exclusive)',那么表示所有的监听器都被触发. 建议先了解"Namespaced event handlers",
不然不明白此处的用意
4034            all = !namespace && !event.exclusive;
4035
4036            /*
调用jQuery.data获取对象缓冲的数据.这些数据有一个标签,就是'events'.很明显这些数据与时间处理有关.事实上,这些数据都是一些事件处理函数.
4037             *
在JavaScript中,"函数也是数据",因此它能够被保存和被修改,以及被传递.也许你对这一个事实并不感到新鲜.
4038             *
这个叫events的数据缓存区的具体位置为jQuery.cache['XXXXXX']['events'].'XXXXXX'表示的是元素的id.这个id由jQuery.data函数来分配
4039             *
并存储在一个叫名字很特别的属性里.这个属性的名称由jQuery.expando指定.具体可以Ctrl+F查找expando的中文注释.
4040             * 如果考虑上event.type,那么下面这句代码的意思就是:
4041             *
到数据缓存区jQuery.cache['XXXXX']['events'][event.type]取出数据,并返回给handlers.如event.type == 'click'
4042             * handlers标明,一个事件类型会有很多的handler,总之不只一个.
4043             */
4044            handlers = ( jQuery.data(this, "events") || {} )[event.type];
4045
4046            //得到了这些事件监听器之后,逐个遍历并执行
4047            for ( var j in handlers ) {
4048                var handler = handlers[j];
4049
4050                // Filter the functions by class
4051                // handler.type
这里这个属性是表示这个handler是哪一个类型的.其实它就是namespace.
在一个监听函数被注册时jQuery.event.add将
4052                // 监听函数的命名空间赋予了监听函数的type属性上.
也就说现在我们看到的handler.type其实就是那名个命名空间.
4053                if ( all || handler.type == namespace ) {
4054                    // Pass in a reference to the handler function itself
4055                    // So that we can later remove it
4056                    //
翻译:给event新建一个属性,这个属性是一个指向handler的引用,这样就方便我们在不需要它时可以删除它.
4057                    event.handler = handler;
4058                    event.data = handler.data;
//保存hanlder缓存的数据,这些数据是在jQuery.event.add函数中被加入的.
4059
4060                    //
this所指的对象是jQuery.event.下面这段代码将hanler作为jQuery.event的方法调用,并将handler的返回值保存到ret中.
4061                    ret = handler.apply( this, arguments );
4062
```

```
4063
                //val在函数的开头被定义,一直到这里才使用.可见val的值一直都是undefined(JS中,变量刚刚比定义的时候值为undefined).
4064                //于是undefined !==false.if内的赋值语句似乎并不会得到执行.我可没忽悠大家啊,大家仔细看好咯.
4065                if ( val !== false )
4066                    val = ret;
4067
4068                // 如果函数执行之后的返回值是false,说明要制止浏览器的默认行为和事件冒泡,调用event中的两个函数完成任务.
4069                //注意,event.prentDefault和event.stopPropagation是w3c定义的方法.IE中要干这两件事情并不是这样的.
4070                //之所以能够统一地以这种方式来调用,这归功于jQuery.event.fix函数.这个函数重写了event.prventDefault和event.stopProgation
4071                if ( ret === false ) {
4072                    event.preventDefault();
4073                    event.stopPropagation();
4074                }
4075            }
4076        }
4077
4078        //返回处理结果,正如你在本函数中看到的,大部分时候这个val是undefined.
4079        return val;
4080    },
4081
4082    /**
4083     *处理各种浏览器(主要是IE和w3c标准)在事件对象模型上的不同.让所有浏览器都能够以一种统一的方式来处理事件对象.
4084     * @param {Event} event 事件对象.
4085     */
4086    fix: function(event) {
4087        //如果event已经有了一个特定的属性并且它的值是true,说明这个事件对象已经被处理过了,直接返回就可以了. 注意,这个属性的名称每刷新一次浏览器都会不一样.另外如果event对象没有被处理过(也可以说没有被标准化过),那event对象是不会有这个特定的属性的.
4088        //
4089        if ( event[expando] == true )
4090            return event;
4091
4092        // store a copy of the original event object
4093        // and "clone" to set read-only properties
4094        var originalEvent = event;
4095        event = { originalEvent: originalEvent };
4096
4097        // 以下这些字符串表示了一个标准的 event 对象所应该具备的标准属性名
4098        var props = "altKey attrChange attrName bubbles button cancelable charCode clientX clientY ctrlKey currentTarget data detail eventPhase fromElement handler keyCode metaKey newValue originalTarget pageX pageY prevValue relatedNode relatedTarget screenX screenY shiftKey srcElement target timeStamp toElement type view wheelDelta which".split(" ");
```

```
4099            for ( var i=props.length; i; i-- )
4100                event[ props[i] ] = originalEvent[ props[i] ];
4101
4102            // Mark it as fixed
4103            // 经过这么处理,event 该有的属性都有了,标记一下
4104            event[expando] = true;
4105
4106
4107
4108            /*
4109             * 属性是有了,但有些属性具体的值却没有啊.
                下面就把这些属性的值补上
4110             *
                在以下的代码中,我们可以见识一下,在事件对象模型方面IE和w3c之间的差异.人家就用这些代码搞定了事件模型的不一致,很值得我们学习.
4111             */
4112
4113
4114
4115            // add preventDefault and stopPropagation since
4116            // they will not work on the clone
4117            /*
                添加preventDefault和stopPropagation两个函数.由于在本函数中让event指向了另外一个新建的对象,原本这两个函数是存在于w3c标准的event
4118             *
                对象中的,现在没了.所以在这里补上,并且改写他们的内容,使得这些函数能够解决兼容性的问题.
4119             */
4120            event.preventDefault = function() {
4121                // if preventDefault exists run it on the original event
4122                //
                w3c标准规定这样设置阻止浏览器默认行为.什么是默认行为?比如说我们按Ctrl+S的时候,是保存网页,点击一个链接的时候会把你导到另一页面等
4123                // 这些都是浏览器的默认行为.
4124                if (originalEvent.preventDefault)
4125                    originalEvent.preventDefault();
4126                // otherwise set the returnValue property of the
                original event to false (IE)
4127                // IE则是用另外一种方式
4128                originalEvent.returnValue = false;
4129            };
4130
4131            /* 接下来是一个停止事件冒泡的函数.
4132             */
4133            event.stopPropagation = function() {
4134                // if stopPropagation exists run it on the original event
4135                // w3c的浏览器这样设置阻止浏览器进行事件冒泡
4136                if (originalEvent.stopPropagation)
4137                    originalEvent.stopPropagation();
4138                // otherwise set the cancelBubble property of the
                original event to true (IE)
4139                //IE则是用另外一种方式
4140                originalEvent.cancelBubble = true;
4141            };
4142
4143            // Fix timeStamp
4144            //
                在IE中是没有'时间戳(timeStamp)'这样东西的.如果没有则给它补上.
```

```
4145            event.timeStamp = event.timeStamp || now();
4146
4147            // Fix target property, if necessary
4148            //
        w3c规定事件的源对象使用target来引用.而在IE中则使用srcElement.
4149            // 这里有一个Safari的bug,官方描述是这样的:"In Safari 2.0
        event.target is null for window load events..."有兴趣可以看:
4150            // http://dev.jquery.com/ticket/1925
        所以为了解决这个问题,我们需要加上" || document
        "来防止event.target为undefined.
4151            if ( !event.target )
4152                event.target = event.srcElement || document; // Fixes
        #1925 where srcElement might not be defined either
4153
4154            // check if target is a textnode (safari)
4155            //
        看样子,safari是允许文本节点捕捉事件的.为了统一行为,让捕捉事件的都是元
        素节点(即要有nodeName),让这些文本节点的容器节点来代替他们捕捉事件
4156            if ( event.target.nodeType == 3/* Node.TEXT_NODE */ )
4157                event.target = event.target.parentNode;
4158
4159            // Add relatedTarget, if necessary
4160            /*
        relatedTarget在w3c事件对象模型中被定义.也就是说,在w3c的事件对象中,应该
        有relatedTarget属性.这个属性与mouseover和
4161             *
        mouseout事件相关,而在其他的事件当中则没用.对于mouseover来说,它是鼠标移
        到目标上时所离开的那个节点.对于mouseout来说,
4162             * 它是离开目标鼠标进入的节点.
4163             *
        在IE中,与relatedTarget对应的就是fromElement和toElement.前者与mouseover
        有关而后者与mouseout有关.
4164             *
        在以下的这语判断中,其实是想判断是不是当前浏览器是不是IE系的浏览器.如果
        是,那么他们是没有relatedTarget但是有fromElement的
4165             */
4166            if ( !event.relatedTarget && event.fromElement )
4167                event.relatedTarget = event.fromElement == event.target ?
     event.toElement : event.fromElement;
4168
4169            // Calculate pageX/Y if missing and clientX/Y available
4170            // 以下这组调整与鼠标定位有关.
4171            /* 事实上,IE与w3c在鼠标相对于视口(view
        port)的定位来说,是兼容的.在这两者的中的事件对象并没有一对叫pageX/pageY
        的属性.
4172             *
        在这里给event添加上这些属性,是为了日后使用的方便(在某些浏览器中似乎有
        这个属性).注意,pageX/pageY说的是鼠标事件发生时
4173             * 鼠标相对于文档位置,他们可以由以下公式计算:
4174             * pageX = clientX + (scrollLef - 边框宽度);
4175             * pageY = clientY + (scrollTop - 边框高度);
4176             *
4177             * scrollLeft是窗口水平滚动量,scrollTop是窗口垂直滚动量.
4178             *
        clientLeft返回边框的宽度clientTop返回边框的高度.真是奇怪怎么他们叫这个
        名字...
4179             */
4180            if ( event.pageX == null && event.clientX != null ) {
```

```
4181            var doc = document.documentElement, body = document.body;
4182            event.pageX = event.clientX + (doc && doc.scrollLeft ||
    body && body.scrollLeft || 0) - (doc.clientLeft || 0);
4183            event.pageY = event.clientY + (doc && doc.scrollTop ||
    body && body.scrollTop || 0) - (doc.clientTop || 0);
4184        }
4185
4186        // Add which for key events
4187        //
    添加一个在键盘事件中非常有用的属性:which.注意这个属性在IE和w3c标准中都
    没有定义.
4188        if ( !event.which && ((event.charCode || event.charCode === 0
    ) ? event.charCode : event.keyCode) )
4189            event.which = event.charCode || event.keyCode;
4190
4191        // Add metaKey to non-Mac browsers (use ctrl for PC's and
    Meta for Macs)
4192        // 在非Macs的机器上,让metaKey就是crtlKey.
4193        if ( !event.metaKey && event.ctrlKey )
4194            event.metaKey = event.ctrlKey;
4195
4196        // Add which for click: 1 == left; 2 == middle; 3 == right
4197        // Note: button is not normalized, so don't use it
4198        //
    COMP:挺恶心的一个问题,那就是鼠标事件中,左中右键键值,也就是button的值的
    问题.我先列个表:
4199        /* browser  left    middle  righ
4200         * IE       1       4       2
4201         * w3c      0       1       2
4202         *
    你可以看到,这些不一致的鼠标键值的定义真的会让人抓狂...下面这段代码就是
    要统一这些定义:
4203         * 1 == left; 2 == middle; 3 == right
4204         *
    下面的嵌套'?:'运算符的确简洁,但是需要各位用心一点研究了.只能意会不能言
    传啊...
4205         */
4206        if ( !event.which && event.button )
4207
    //注意这里是一个'&'运算符,而不是'&&'
4208            event.which = (event.button & 1 ? 1 : ( event.button & 2
    ? 3 : ( event.button & 4 ? 2 : 0 ) ));
4209
4210        return event;
4211    },
4212
4213    /**
4214     * 将fn用proxy包装起来, 最后返回proxy.
4215     * 从函数内容来看,
    这个函数的作用仅仅就是将fn的唯一编号(guid)赋给proxy而已
4216     * @param {Function} fn - 元素的函数,它有一个guid
4217     * @param {Function} proxy - 最后将返回这个函数,
    返回时它将具有fn的guid. 一般来说, proxy仅仅是fn的一层包裹.
4218     */
4219    proxy: function( fn, proxy ){
4220        // Set the guid of unique handler to the same of original
    handler, so it can be removed
4221        //通过下面的这句代码确保proxy具有和fn一样的guid,
```

```
从而它能够被移除.
4222            proxy.guid = fn.guid = fn.guid || proxy.guid || this.guid++;
4223            // So proxy can be declared as an argument
4224            return proxy;
4225        },
4226
4227        /**
4228         * spacial 是一个键/值集合,它记录了所谓的 " 特殊事件 ".
4229         * "特殊事件"有三个,分两类:
4230         *
第一类特殊事件并不是浏览器所支持的,在这里就是ready事件.也就是说并没有
哪一个浏览器有onReady这样的事件.其实ready取DOM ready之意, 部分浏览器有
4231         * DOMContentLoaded事件,但是并不是普及.
jQuery通过使用自创的ready事件做为代理(delegate),
根据当前浏览器进行区别对待:(1)如果是支持DOMContentLoaded
4232         * 事件的浏览器, 就直接将其绑定. (2)如果是不支持该事件的浏览器,
则通过模拟的方式来"绑定"事件. 具体参见jQuery.event.bindReady函数.
4233         *
4234         * 第二类是"异曲同工"的事件,在这里就是mouseenter和mouseleave.
mouseenter和mouseleave是IE事件模型中的两个经典鼠标事件,
w3c与之相对应的事件是我们更加
4235         * 熟悉的mouseover与mouseout.
如果程序员在非IE浏览器中要求绑定事件处理函数到mouseenter/mouseleave事
件, 则jQuery会绑上与之相对应的w3c事件.
4236         */
4237        special: {
4238            ready: {
4239                /**
4240                 * 经过上下文的分析, setup也是"绑定"之意, 下同.
4241                 * 此函数用来完成ready事件的响应函数的绑定. 再次提醒,
浏览器中并没有ready事件, ready是jQuery "原创"的事件.
4242                 */
4243                setup: function() {
4244                    // Make sure the ready event is setup
4245                    bindReady();//其实是调用了bindReady函数进行绑定.
4246                    return;//函数就这么return了, 这个时候的返回值!==
false, 注意啊.
4247                },
4248
4249                /*
4250                 *
卸载这个事件.由于ready事件并不是通过"正规"的途径(addEventListener/atta
chEvent)绑定的, 因此当然使用"非常"的方式来进行卸载.
4251                 */
4252                teardown: function() { return; }
4253            },
4254
4255
4256            mouseenter: {
4257                /**
4258                 *
如果是在IE中使用setup来绑定这个事件,则不使用setup来绑定,因为有更简便的
方式(attachEvent).
4259                 * 如果在非IE的浏览器中要求绑定mouseenter事件, 那不管,
统一给它绑定mouseover这个标准的事件.
4260                 *
4261                 * 注意本函数内的this关键字的指向,
由于setup方法是在jQuery.event.add中这样被使用的:
```

```
4262                 * if ( !jQuery.event.special[type] ||
    jQuery.event.special[type].setup.call(elem) === false )//...other
    codes
4263                 * 所以,
    setup方法中的this指的是就是上面那行代码中的elem.
    而elem就是一个普通的DOM 元素.
4264                 */
4265                setup: function() {
4266                    if ( jQuery.browser.msie ) return false;
4267
    //这个this关键子指向是一个普通的DOM元素,也就是说我们要给这个元素绑定事
    件处理函数
4268                    jQuery(this).bind("mouseover", jQuery.event.special.
    mouseenter.handler);
4269                    return true;
4270                },
4271                /**
4272                 * mouseenter事件的卸载函数.
4273                 */
4274                teardown: function() {
4275                    if ( jQuery.browser.msie ) return false;
    //如果是IE浏览器有更好的卸载方法,返回false,
    这样调用teardown的函数就知道直接使用
4276
    //IE的detachEvent来卸载mouseenter
4277                    jQuery(this).unbind("mouseover", jQuery.event.special
    .mouseenter.handler);
4278                    return true;
4279                },
4280
4281                handler: function(event) {
4282                    // If we actually just moused on to a sub-element,
    ignore it
4283                    // 翻译:如果我们仅仅是移动到了元素的子元素上,
    忽略它(并不把这种情况当作是鼠标enter到了元素上).
4284                    if ( withinElement(event, this) ) return true;
4285
4286                    // Execute the right handlers by setting the event
    type to mouseenter
4287                    // 将event的事件类型改为mouseenter
4288                    event.type = "mouseenter";
4289
    //TODO:这个this指代的是一个什么对象?
4290                    return jQuery.event.handle.apply(this, arguments);
4291                }
4292            },
4293
4294            /**
4295             * mouseleave的注释参见mouseenter的中文注释
4296             */
4297            mouseleave: {
4298
4299                setup: function() {
4300                    if ( jQuery.browser.msie ) return false;
4301                    jQuery(this).bind("mouseout", jQuery.event.special.
    mouseleave.handler);
4302                    return true;
4303                },
```

```
4304
4305            teardown: function() {
4306                if ( jQuery.browser.msie ) return false;
4307                jQuery(this).unbind("mouseout", jQuery.event.special.
    mouseleave.handler);
4308                return true;
4309            },
4310
4311            handler: function(event) {
4312                // If we actually just moused on to a sub-element,
    ignore it
4313                if ( withinElement(event, this) ) return true;
4314                // Execute the right handlers by setting the event
    type to mouseleave
4315                event.type = "mouseleave";
4316                return jQuery.event.handle.apply(this, arguments);
4317            }
4318        }
4319    }
4320 };//完成静态事件方法的定义.
4321
4322
4323                    /*
4324                     *
    下面就给jQuery对象添加事件相关的方法.其实质是jQuery静态事件方法的
4325                     *
    封装.完成事件方面的任务时,'幕后黑手'主要仍然是上面定义的静态方法.
4326                     */
4327
4328 // ------------------------------------------------下面给 jQuery
    对象添加事件方面的方法
    ------------------------------------------------------------
4329
4330 /**
4331  * 使用jQuery对象的extend函数还为jQuery对象扩展事件方面的方法.
4332  */
4333 jQuery.fn.extend({
4334     /**
4335      * 为jQuery对象中的匹配元素集合绑定事件处理函数:
4336      *
    如果是事件类型是unload就是调用jQuery对象自己的one函数来为匹配元素集合
    中的每一个元素在unload事件上绑定一个只执行一次的监听函数.
4337      *
4338      * 如果不是,就遍历jQuery对象匹配元素集合内的每一个元素,
    并为每一个元素绑定传入进来的事件处理函数.
4339      * 请注意本函数内的this关键的具体含义,比较混乱,请保持清醒...
4340      *
4341      * @param {string} type - 事件类型, 如"click","mouseenter"等
4342      * @param {Array} data - 需要绑定到事件处理函数上的数据.
    有时候bind方法只有两个参数, 即没有传入下面那个参数,
    那就把这个参数作为处理函数.
4343      * @param {Function} fn - 事件处理函数
4344      */
4345     bind: function( type, data, fn ) {
4346                                     //这个this是jQuery对象
    //这个也是
4347         return type == "unload" ? this.one(type, data, fn) : this.
    each(function(){
```

```
4348
                //这个this指的是匹配元素集合内的每一个元素
4349                jQuery.event.add( this, type, fn || data, fn && data );
4350            });
4351        },
4352        /**
4353         *
                为匹配元素集合中的每一个元素为type所指定的事件绑定个一次性的函数.
                这些函数只被执行一次.其使用方法同jQuery.fn.bind;
4354         *
4355         * @param {string} type - 事件类型, 如"click"
4356         * @param {Array} data - 需要绑定到事件处理函数上的数据.
                有时候bind方法只有两个参数, 即没有传入下面那个参数,
                那就把这个参数作为处理函数.
4357         * @param {Function} fn - 事件处理函数
4358         */
4359        one: function( type, data, fn ) {
4360
4361
                //proxy函数将为匿名函数"安装"上一个guid属性.这个属性的值跟fn或data是一样的. proxy函数应用的场合一般是想扩展
4362                                        //事件处理函数,
                在事件处理函数之前或之后再添加一些操作.这里定义了一个one函数,
                它的作用是把监听函数装起来,然后在
4363
                //监听函数执行之前把这个函数从当前元素上卸载.
4364            var one = jQuery.event.proxy( fn || data, function(event) {
4365                //注意, this指的是当前正在处理的匹配元素集合中的元素.
4366                jQuery(this).unbind(event, one);//首先卸载监听函数one
4367                return (fn || data).apply( this, arguments );
                //然后执行真正的fn
4368            });
4369            return this.each(function(){//好,
                为匹配元素集合中的每一个元素绑定绑定one这个监听函数.
4370                jQuery.event.add( this, type, one, fn && data);
4371            });
4372        },
4373
4374        /**
4375         * 为jQuery对象卸载指定事件类型上的指定监听函数
4376         * @param {string} type - 卸载的事件类型
4377         * @param {Function} fn - 需要卸载的函数的引用
4378         */
4379        unbind: function( type, fn ) {
4380            return this.each(function(){
4381
                //this指的是匹配元素集合中的每一个元素
4382                jQuery.event.remove( this, type, fn );
4383            });
4384        },
4385        /**
4386         * 触发绑定在type所指定的事件上的监听函数.
                匹配元素集合中的每一个元素的事件监听函数都触发.
4387         * @param {string} type - 所要触发的事件类型
4388         * @param {Array} data - 需要传给事件监听函数的参数
4389         * @param {Function} fn - 触发监听函数运行之后,
                你需要再执行的一些操作.
4390         */
```

```
4391        trigger: function( type, data, fn ) {
4392            return this.each(function(){
4393                jQuery.event.trigger( type, data, this, true, fn );
4394            });
4395        },
4396        /**
4397         * 仅触发绑定在匹配元素集合第一个元素上,
    并且是type所指定的事件上的监听函数.
4398         * @param {string} type - 所要触发的事件类型
4399         * @param {Array} data - 需要传给事件监听函数的参数
4400         * @param {Function} fn - 监听函数运行之后,
    你需要再执行的一些操作.
4401         */
4402        triggerHandler: function( type, data, fn ) {
4403            return this[0] && jQuery.event.trigger( type, data, this[0],
    false, fn );
4404        },
4405        /**
4406         * 来自API文档的摘抄:
4407         * 每次点击后依次调用函数。
4408         *
    如果点击了一个匹配的元素，则触发指定的第一个函数，当再次点击同一元素时
    ，则触发指定的第二个函数，如果有更多函数，则再次触发，直到最后一个。
4409         * 随后的每次点击都重复对这几个函数的轮番调用。
4410         * 可以使用unbind("click")来删除。
4411         *
4412         * 好,我来点说明:
4413         * (1) 本函数需要接收两个以上的函数引用作为参数, 这里只有一个,
    请注意;
4414         * (2) 第2个之后的所有函数拥有与第1个函数一样的函数ID.
4415         *
4416         * @param {Function} fn - 需要切换的函数, 可以传入多个Function.
4417         */
4418        toggle: function( fn ) {
4419            // Save reference to arguments for access in closure
4420            // 翻译: 保存arguments的引用, 这样待会在闭包中还能访问到它.
4421            var args = arguments, i = 1;
4422
4423            // link all the functions, so any of them can unbind this
    click handler
4424            // 所有函数都使用给第1个函数一样的函数ID,
    这样可以在卸载时统一删除.
4425            while( i < args.length )
4426                jQuery.event.proxy( fn, args[i++] );
    //proxy函数仅仅为第二个参数添加一个与第一个参数一样的函数ID.
4427
4428            //添加click事件的监听函数.
    在监听事件里面轮流调用args内的每一个函数. 注意匿名函数中的i,
    它在上面那个函数运行完毕之后等于args的长度了.
4429            return this.click( jQuery.event.proxy( fn, function(event) {
4430
4431                /*
    注意本匿名函数中的this指向的是当前正在处理的匹配元素集合中的那一个元素
     */
4432
4433                // Figure out which function to execute
4434                // 翻译: 计算出需要运行哪一个函数
4435                this.lastToggle = ( this.lastToggle || 0 ) % i;
```

```
4436
4437            // Make sure that clicks stop
4438            event.preventDefault();//取消浏览器的默认行为.
4439
4440            // and execute the function 翻译:运行这个函数
4441            return args[ this.lastToggle++ ].apply( this, arguments )
 || false;
4442        }));
4443    },
4444
4445    /**
4446     * 分别注册两个监听函数到鼠标移入和移出两个事件上.
4447     * @param {Function} fnOver
4448     * @param {Function} fnOut
4449     */
4450    hover: function(fnOver, fnOut) {
4451        return this.bind('mouseenter', fnOver).bind('mouseleave',
fnOut);
4452    },
4453    /**
4454     * 将指定的一个fn绑定到DOM Ready事件发生时执行.
4455     *
4456     * @param {Function} fn - ready的监听事件.当DOM Ready时,
此函数将会被调用
4457     */
4458    ready: function(fn) {
4459        // Attach the listeners
4460        bindReady();//将jQuery.ready函数绑定到DOM
Ready(也即DOMContentLoaded)时执行.
4461
4462        // If the DOM is already ready
4463        // 如果这个时候DOM已经ready了, 那事不宜迟,
马上执行.将fn作为document的方法调用,并将jQuery的构造函数作为参数传入,
方便fn进行处理.
4464        if ( jQuery.isReady )
4465            // Execute the function immediately
4466            fn.call( document, jQuery );
4467
4468        // Otherwise, remember the function for later
4469        // 否则,DOM 还没ready,
那就把事件监听函数放入一个readyList当中,待到DOM
Ready事件真的发生了, 那么jQuery.ready函数将会被执行.jQuery.ready
4470        // 函数则遍历readyList中的等待函数, 逐个执行它们.
4471        else
4472            // Add the function to the wait list 翻译:
将函数加入到等待队列当中
4473            /* 这里有一个问题值得思考,
为什么不直接把fn放入readyList当中, 而是要在它外边再包裹多一层呢?
4474             * 正如我前面所说的, 这种 "包裹方法"
的做法在jQuery中是想改变fn的调用上下文以及传递参数.通过对jQuery.ready
代码的分析,我们发现readyList
4475             *
中的每一个函数都会被这样调用:"this.call(document)",注意this在该环境中
指的是当前正在处理的readyList中的函数. 这样下面的这个包裹的
4476             * 函数就会被作为document的一个方法来执行,
于是下面这个包裹函数体中的this指的就是document.
4477             */
4478            jQuery.readyList.push( function() {
```

```
4479                    return fn.call(this, jQuery);
4480                });
4481
4482            return this;
4483        }
4484    });
4485
4486    //
        --------------------------------------------------------------------
        ----------------------------------------------------------------
4487
4488
4489
4490
4491
4492
4493
4494    // ------------------------------------ jQuery DOM Ready方面的
        静态方法. 这也是事件模块的一部分
        ------------------------------------------------------------
4495
4496    jQuery.extend({
4497        isReady: false,// 让jQuery获得isReady这个属性,
        这个属性初始为false说明DOM结构还没建立.注意isReady表示的时机与window.o
        nload是不一样的
4498                        // 具体参照bindReady的中文注释.
4499        readyList: [],//等待DOM Ready事件的监听函数列表.
4500
4501        // Handle when the DOM is ready
4502        /**
4503         * 当DOM Ready之后, 这个函数马上就会被执行.
        而这个函数就会逐个执行readyList中的函数.
        readyList中函数就是绑定到DOM Ready事件上的函数.
4504         * DOM
        Ready是文档结构生成完毕,但是内容尚未加载完毕时(如HTML文档生产完毕,
        但是图片内容尚未加载完毕.)
4505         */
4506        ready: function() {
4507            // Make sure that the DOM is not already loaded
4508            if ( !jQuery.isReady ) {
4509                // Remember that the DOM is ready
4510                jQuery.isReady = true;
4511
4512                // If there are functions bound, to execute
4513                if ( jQuery.readyList ) {
4514                    // Execute all of them
4515                    // 执行所有绑定到DOM Ready事件上的函数,
        这些函数都被装在readyList当中.
4516                    //
        jQuery.each函数遍历readyList当中每一个函数,并用document作为这些函数的
        上下文而执行他们.
4517                    jQuery.each( jQuery.readyList, function(){
4518                        // 把 readyList
        里面的每一个函数都作为document的方法运行
        .注意this关键字指的是readyList中当前正在执行的监听函数
4519                        this.call( document );
4520                    });
4521
```

```
4522                    // Reset the list of functions
4523                    // 执行完毕就将 readyList 清空
4524                    jQuery.readyList = null;
4525                }
4526
4527                // Trigger any bound ready events
4528                jQuery(document).triggerHandler("ready");
4529            }
4530        }
4531    });
4532
4533    //
        ------------------------------------------------------------------------
        ------------------------------------------------
4534
4535
4536    var readyBound = false;
        //初始化的时候,我们认为还没有将事件监听函数绑定到DOM Ready事件当中.
        只要bindReady运行完毕, 这个变量就会变成true, 表示
4537                            //DOM
        Ready事件已经模拟完毕(注意没有浏览器支持DOM Ready事件,
        有部分支持DOMContentLoaded),并且jQuery.ready函数已经
4538                            //绑定到该事件上.
4539
4540
4541
4542
4543    /**
4544     * 将jQuery.ready绑定到"DOMContentLoaded"事件中执行.
4545     * "DOMContentLoaded"事件目前并没有得到普及,
        只有下面所述的Mozilla,Opera等浏览器实现这个事件.这个事件是指文档的HTML
        框架创建好的那一个时刻,
4546     * 比如说HTML框架渲染完毕, 但是图片还没有load回来之前.
4547     * 使用这个事件的好处是文档结构一建立,
        就马上可以执行JavaScript代码, 而不用等到大量的图片加载完毕才执行,
        增强了用户体验.
4548     */
4549    function bindReady(){
4550        if ( readyBound ) return;
4551        readyBound = true;
4552
4553        // Mozilla, Opera (see further below for it) and webkit
        nightlies currently support this event
4554        if ( document.addEventListener && !jQuery.browser.opera)
4555            // Use the handy event callback
4556            document.addEventListener( "DOMContentLoaded", jQuery.ready,
        false );
4557
4558        /*
4559         * 由于IE 和 Safari不支持"DOMContentLoaded"事件,
        因此需要下面的代码来模拟这个事件.
4560         * 而Opera虽然支持, 但是支持的方式比较特别,
        因此也需要另外的代码来绑定jQuery.ready到这个事件.
4561         *
4562         * COMP:
4563         * 以下代码显示了各种浏览器检查DOM
        Ready(即DOMContentLoaded)不同方式
4564         */
```

```
4565
4566        // If IE is used and is not in a frame
4567        // Continually check to see if the document is ready
4568        /*
4569         * 翻译: 如果当前的浏览器是IE并且当前页面不在一个框架当中,
            就不断地检测文档是否准备就绪.
4570         */
4571        if ( jQuery.browser.msie && window == top ) (function(){//
            如果一个页面的处在一个框架当中(in frame),
            那么它有一个全局的变量top指向
4572            if (jQuery.isReady) return;                          //
            指向这个框架的顶层框架的windwo对象. 因此window == top 就为false.
4573            try {
4574                // If IE is used, use the trick by Diego Perini
4575                // http://javascript.nwbox.com/IEContentLoaded/
4576                document.documentElement.doScroll("left");//
            IE中使用这个方法检测DOM是否Ready,
            正如Resig所说的,这是一个trick(花招)有兴
4577                                                          //
            趣可深究没兴趣没时间记住了也可以.
4578            } catch( error ) {//发生错误说明IE中DOM没Ready, 在catch中重试!
4579                setTimeout( arguments.callee, 0 );
            //agruments.callee是一个函数本身的引用,
            而在0毫秒后重试是有一定的技巧的:
4580                                                  /*
            函数延迟0毫秒执行并不是立即执行,
            而是等浏览器运行完挂起的事件句柄和已经更新完文档状态之后才
4581                                                  *
            运行这个函数.详情见《JavaScript Definition Guide 5th
            Edition(JavaScript权威指南第5版)》
4582                                                  */
4583                return;//设置完定时器之后, 返回.
4584            }
4585            // and execute any waiting functions
4586            jQuery.ready();//当首次测试到DOM Ready之后,
            马上运行jQuery.ready函数, 而jQuery.ready函数又会运行所有绑定到DOM
            Ready事件上的函数
4587
            //这些函数的引用被存储在jQuery.readyList当中.
            可以参考jQuery.ready的中文注释.
4588        })();
4589
4590        // Opera浏览器支持"DOMContentLoaded"事件,
            但是这个DOMContentLoaded事件是不算上stylesheet的,即如果stylesheet文件
            没有链接完毕,但DOM OK了,
4591        // Opera也算这个时刻为"DOMContentLoaded".
            我们的js代码有可能会需要获取stylesheet里面的值然后做出相应的动作,但是
            在stylesheet还没enable
4592        // 之前做这些动作是不安全的. 因此,
            下面的代码将jQuery.ready函数再包装一层,
            在执行jQuery.ready之前确保stylesheet是enable的.
4593        if ( jQuery.browser.opera )
4594            document.addEventListener( "DOMContentLoaded", function () {
4595                if (jQuery.isReady) return;
4596
            //逐个检查stylesheet是不是enable的.注意,这个stylesheet包括link进来的和
            嵌入式(<style>标签里面)的.
4597                for (var i = 0; i < document.styleSheets.length; i++)
```

```
4598                if (document.styleSheets[i].disabled) {
    //一旦检测到有disabled的stylesheet,
    就需要重新执行这个(bindReady)函数了.
4599                    setTimeout( arguments.callee, 0 );//
    设置bindReady函数在0毫秒后执行.
4600                    return;
4601                }
4602            // and execute any waiting functions
4603            // 如果上面的for循环顺利的话,
    即没有碰到disabled的stylesheet, 那就不会被return截断函数的执行,
    这个时候可以执行jQuery.ready了.
4604            jQuery.ready();
4605        }, false);//false表示不需要在事件的捕捉阶段处理事件.
4606
4607    // safari不支持"DOMContentLoaded"事件, 于是需要采取措施来模拟:
4608    if ( jQuery.browser.safari ) {
4609        var numStyles;//用来记录文档中链接的stylesheet的数目.
    Safari将用它来检测stylesheet是否全部加载完毕.
4610        (function(){
4611            if (jQuery.isReady) return;
4612
4613            /*
4614             * 由于safari没有IE的"doScroll"的trick,
    因此就比较麻烦了,需要通过readyState来判断
4615             */
4616
4617            //"loaded" 和 "complete" 是我们想要的状态,
    如果不是这两个状态,则重新执行bindReady
4618            if ( document.readyState != "loaded" && document.
    readyState != "complete" ) {
4619                /*
    readyState有5种状态(下表只是一种解释,可以看到,还有不少其他的解释):
4620                 * 0 = uninitialized    未初始化,还没发送请求
4621                 * 1 = loading          正在发送请求
4622                 * 2 = loaded
    请求发送完毕,已经接收到全部的响应内容
4623                 * 3 = interactive      正在解析响应内容
4624                 * 4 = complete         响应内容解析完成
4625                 */
4626                setTimeout( arguments.callee, 0 );
    //延迟0微秒(具体作用可以看上面的注释)之后,再重新执行bindReady.
4627                return;
4628            }
4629
4630            /*
4631             * 经过上面那个if,
    并不能保证就是我们所要的"DOMContentLoaded",
    因为这个时候也同样面临着跟Opera一样的stylesheet可能未加载完毕
4632             * 的危险.下面的代码就是要解除这个危险.
4633             */
4634
4635            if ( numStyles === undefined )// 一个变量初始化的时候,
    如果没复制, 就是"undefined"
4636                numStyles = jQuery("style, link[rel=stylesheet]").
    length;// 注意,这里有两个选择器
4637            if ( document.styleSheets.length != numStyles ) {
    //数目不对表示stylesheet没有加载完成, 重新执行bindReady
4638                setTimeout( arguments.callee, 0 );
```

```
4639                    return;
4640                }
4641                // and execute any waiting functions
4642                // 好了, 代码如果能运行到这里, 那么一切就绪了, 可以执行jQuery.ready!
4643                jQuery.ready();
4644            })();
4645        }
4646
4647        // A fallback to window.onload, that will always work
4648        // 将 jQuery.ready 方法绑定到load 事件.这样不管浏览器是否支持DOM Ready, jQuery.ready内的代码总是被执行, 慢就慢点咯.
4649        // 或许有人会问, 那jQuery.ready岂不是执行了两次? 不会,因为jQuery.ready函数会检查标志变量的isReady的. 当jQuery.ready
4650        // 函数被执行过一次之后, isReady就为true了. 具体参见jQuery.ready的代码.
4651        jQuery.event.add( window, "load", jQuery.ready );
4652    }
4653
4654
4655
4656    //--------------------------------------------------------------------
        -- jQuery 对象对各种事件的绑定函数
        ------------------------------------
4657
4658    //为"blur,focus,load,resize,scroll,unload,click,dblclick..."等事件定义一个事件绑定函数.
4659    jQuery.each( ("blur,focus,load,resize,scroll,unload,click,dblclick," +
4660        "mousedown,mouseup,mousemove,mouseover,mouseout,change,select," +
4661        "submit,keydown,keypress,keyup,error").split(","), function(i, name){
4662
4663        /*
4664         * 为了说明问题, 以下注释将举一个例子来说明到底这段代码干了些什么事情. 假设
            现在的name = 'click', 于是 i=6 (i是click在split产生的数组中的排序)
4665         */
4666
4667        // Handle event binding
4668        // 经过上面的假设, 现在name = 'click', 于是你可看到, 下面的代码在jQuery对象上定义了一个叫"click"的方法. 它有一个参数, 这个参数就是你需要在
4669        // HTML元素被click时的响应函数, 也即监听函数. 如果这个函数没有被传入click方法, 那么jQuery对象就会触发所有绑定在click事件上的所有监听函数.
4670        jQuery.fn[name] = function(fn){
4671            return fn ? this.bind(name, fn) : this.trigger(name);
4672        };
4673    });
4674
4675    // Checks if an event happened on an element within another element
4676    // Used in jQuery.event.special.mouseenter and mouseleave handlers
4677    /**
4678     * 意译上面那段英文:检测鼠标mouseenter事件和mouseleave事件发生时, 鼠标是不是真的在事发元素上, 而不是在它的子元素上.
4679     * 这个函数在 Query.event.special.mouseenter 和 mouseleave 的handler中被使用.
        当这个函数判断到鼠标处在元素的子元素上时,handler函数就
```

```
4680     * 会return.
4681     * @param {Event} event - 事件对象
4682     * @param {HTMLElement}  elem - 检测到底是不是在这个元素里面
4683     */
4684    var withinElement = function(event, elem) {
4685        // Check if mouse(over|out) are still within the same parent
         element
4686        /* relatedTarget对于mouseover来说,
         它是鼠标移动到目标元素上时所离开的那个元素; 而对于mouseout来说,
         它是离开目标时,鼠标进入的那个元素.
4687         * 详情参见《JavaScript Definition Guide 5th Edition(
         JavaScript 权威指南 )》
4688         */
4689        var parent = event.relatedTarget;
4690        // Traverse up the tree
4691
4692        //
         一直往上查找看看鼠标的当前关联元素的父亲或者祖先是不是函数第二个参数所
         指定的那个元素(elem), 是就说明还在elem元素上.
4693        while ( parent && parent != elem ) try { parent = parent.
         parentNode; } catch(error) { parent = elem; /*
         发生错误也认为鼠标仍在元素内 */}
4694        // Return true if we actually just moused on to a sub-element
4695        return parent == elem;
4696    };
4697
4698    // Prevent memory leaks in IE
4699    // And prevent errors on refresh with events like mouseover in other
         browsers
4700    // Window isn't included so as not to unbind existing unload events
4701    // 翻译: 避免IE中内存泄漏.
4702    // 并且避免在其他浏览器中像mouseover那样的事件刷新时所导致的错误
4703    // Window对象并没有包括在内,
         以免将已经绑定在window.unload事件上的监听函数卸载掉(因为整个unload过程
         由window.unload触发, 你删了它, 那还怎么触发啊)
4704    jQuery(window).bind("unload", function() {
4705
         //所有包括document在内的文档元素在unload事件发生时卸载掉所有的事件监听
         函数.
4706        jQuery("*").add(document).unbind();
4707    });
4708
4709
4710
4711
4712    //------------------------------------------------------------------
         扩展 jQuery对象使其具有 ajax方面的能力 ------------------------
4713    jQuery.fn.extend({
4714        // Keep a copy of the old load
4715        _load: jQuery.fn.load,//将原来老的load方法保存.
4716
4717        /**
4718         *  让一个 jQuery
         对象获得加载远程文档并且将加载的内容放到自己里边.
         这个函数使用了jQuery 有关ajax 方面的静态那函数
4719         *
4720         * @param {string} url
4721         * @param {Object} params
```

```
4722        * @param {Function} callback
4723        */
4724       load: function( url, params, callback ) {
4725           if ( typeof url != 'string' )
4726               return this._load( url );
4727
4728           // 看看url 里面是否含有selector, 有selector,
       说明并不想把请求页面的所有内容都load回来,只是想load selector指定的部分
4729           var off = url.indexOf(" ");
4730           // off >= 0, 说明含有selector, 分离出真正的url和selector
4731           if ( off >= 0 ) {
4732               var selector = url.slice(off, url.length);
4733               url = url.slice(0, off);
4734           }
4735
4736           //保证要有一个回调函数
4737           callback = callback || function(){};
4738
4739           // Default to a GET request load缺省使用GET方法获取数据
4740           var type = "GET";
4741
4742           // If the second parameter was provided
4743           // 如果提供了第二个参数,那就根据这个参数的类型做出一些调整
4744           if ( params )
4745               // If it's a function
4746               // 如果params是函数,
       那就对load函数的各个参数进行"矫正"处理:
4747               if ( jQuery.isFunction( params ) ) {
4748                   // We assume that it's the callback
4749                   callback = params;
4750                   params = null;
4751
4752               // Otherwise, build a param string
4753               // 否则,
       使用jQuery.param方法将params对象转化为一个字符串.这个过程叫做"参数串行
       化"
4754               } else {
4755                   params = jQuery.param( params );
       //jQuery.param将params对象转化成为字符串,
       以便load方法使用POST方式将这些参数传输到服务器端
4756                   type = "POST";//如果有参数传入,
       那我们就使用POST方法传递参数并获取结果.
4757               }
4758
4759           var self = this;//保存this的引用,
       因为接下来的函数定义中this的含义将会被改变.
4760
4761           // Request the remote document
4762           //
       使用静态的ajax方法请求远程的文档.ajax函数接收一个对象作为参数,
       这个对象对本次ajax请求进行了设置:
4763           jQuery.ajax({
4764               url: url,//从这个url加载数据
4765               type: type,//请求的类型,GET or POST or Others
4766               dataType: "html",// 将请求到的数据当作HTML来解析
4767               data: params,// 发送到服务器端的参数, 如果它有值,
       则type将会被改为POST, 因为GET方法不适宜发送大量数据
4768               /**
```

```
4769                * 当请求成功时所调用的函数,
    可以是它是jQuery的"系统级"回调,
    一般情况下jQuery的应用级程序员(也就是使用jQuery的程序员)不需要关注这个
    函数
4770                * 他们只需要关注自己"应用级callback",
    即自己写的callback便可.
4771                *
4772                *
    也许你觉得为什么就是这两个参数而不是其他的参数?能不能修改?
4773                * 要解决你这个问题,
    你必须查看OnComplete事件监听器的代码,
    是它决定了complete函数所能获得的参数.
4774                *
4775                * @param {Object} res - 服务器返回的内容
4776                * @param {Object} status - 响应的状态
4777                */
4778               complete: function(res, status){
4779                   // If successful, inject the HTML into all the
    matched elements
4780                   // 翻译: 如果请求成功, 将HTML注入到所有匹配元素当中去.
4781                   if ( status == "success" || status == "notmodified" )
4782                       // See if a selector was specified
4783                       // 如果请求url中有选择器,
    那么在请求结果中过滤出选择器所指定的部分, 然后再注入.
    如url为"info.php #news", 则仅将请求回来的页面
4784                       // 中#news里的HTML注入到目标元素当中.
4785                       self.html( selector ?
4786                           // Create a dummy div to hold the results
4787                           // 把请求回来的内容先注入到一个空壳div中
4788                           jQuery("<div/>")
4789                               // inject the contents of the document
    in, removing the scripts
4790                               // to avoid any 'Permission Denied'
    errors in IE
4791                               /* 翻译: 注入整个文档的内容,
    与此同时去除调scripts标签以免在IE中导致 "Permission Denied" 的错误
4792                                */
4793                               .append(res.responseText.replace(
    /<script(.|\s)*?\/script>/g, ""))
4794
4795                               // Locate the specified elements
4796                               // 翻译: 定位指定的元素.
    其实就是过滤出selector所指定的内容最后作为selt.html的参数
4797                               .find(selector) :
4798
4799                           // If not, just inject the full result
4800                           // 如果没有指定选择器, 那好办,
    直接把内容作为self.html的参数,
    直接注入到匹配元素集合中的每一个元素中去.
4801                           res.responseText );
4802                   // 在匹配元素集合中的每一个元素上调用callback函数,
    并以[res.responseText, status, res] 作为参数
4803                   self.each( callback, [res.responseText, status, res]
    );
4804               }
4805           });
4806           return this;//返回jQuery对象的引用, 方便链式调用.
4807       },
```

```
4808
4809        /**
4810         * 串行化参数.
            将jQuery对象上的需要被串行化的参数对象转化为一个类似"a=value1&b=value2
            &c=value3"的字符串, 方便GET(参数不多时)或者POST使用.
4811         * 可以看到, 这个serialize不过是一个Facade(门面),
            实际完成工作的是jQuery.param函数.
            具体可以参考jQuery.param函数的中文注释.
4812         */
4813        serialize: function() {
4814            return jQuery.param(this.serializeArray());
4815        },
4816        /**
4817         *
            将jQuery对象中匹配元素集合中的元素转换成一个数组.数组中的每一个元素是
            一个对象, 每个对象的结果如下:
4818         * {name:"name",value:"value"}
4819         */
4820        serializeArray: function() {
4821                        //
            map函数对匹配元素集合内的每一个元素调用参数中的那个函数进行处理,
            并用这个处理结果新建一个jQuery对象, 并用这个新对象替换原来的
4822                        // 旧jQuery对象,
            最终map函数将返回这个新jQuery对象的引用, 从而方便方法的链式调用.
4823            return this.map(function(){
4824
4825                //
                注意以下代码中的this指的是匹配元素集合中当前正在处理的的那个元素
4826
4827                // 看看当前这个元素是不是form,
                如果是就将form内的元素(elemsts)使用jQuery.makeArray转化为数组返回(因为
                form内的字段才是我们需要的);
4828                // 如果不是那就直接元素返回.
                可见serializeArray函数中的map的作用主要是针对form元素的.
4829                return jQuery.nodeName(this, "form") ?
4830                    jQuery.makeArray(this.elements) : this;
4831            })
4832            /*
4833             * filter 函数筛选出符合条件的元素.而筛选条件就由传入
                filter的方法( fn )来决定.这个方法( fn ) retrun
                回来的元素就是要过滤出来的元素.
4834             */
4835            .filter(function(){
4836                //
                把表单中的有name属性的,可见的,能用的,被选上的,总之是要传到服务器端的参
                数全部留下. 注意这个过滤函数的返回值是true/false, true就是
4837                // 将当前正在过滤的这个元素留下, false的话就是不要.
4838                return this.name && !this.disabled &&
4839                    (this.checked || /select|textarea/i.test(this.
                nodeName) ||
4840                        /text|hidden|password/i.test(this.type));
4841            })
4842            /*
4843             * 将筛选出来的元素最后map过一次之后重新组合成一个数组.
                这个最后一次的map主要是HTML元素转化为一个具有"name"和"value"的对象.
4844             */
4845            .map(function(i, elem){
4846                                        // 使用val()
```

```
        函数取得第一个匹配元素标签内的值(jQuery(this)也就一个匹配元素),有点innerText的味道...
4847            var val = jQuery(this).val();
4848            // "? :" 运算符总是那么简洁, 比较难用语言表述.
4849            return val == null ? null :
4850                val.constructor == Array ?//如果val.constructor是Array,则this就是一个类似Select那样的元素,能够选择多个值,所以valu()才会返回Array
4851                    jQuery.map( val, function(val, i){//如果val是数组, 那就将数组里面的每一个元素映射为{name: elem.name, value: val}.
4852                                                      //这个val是一个数组
4853                        return {name: elem.name, value: val};
4854                    }) :
4855                                                  // 这个val 是一个字符串
4856                    {name: elem.name, value: val};
4857        })
4858        .get();/*
4859                  * get()方法将jQuery对象的匹配元素集合"抽"出来,得到一个数组, 并将这个数组返回.
4860                  * 如果你不喜欢操作jQuery对象(这个对象里面包含选择器所选择到的所有元素 ), 那么调用jQuery 对象的get 函数将会获
4861                  * 得一个由匹配元素集合里元素所组成的数组,这样你就可以直接操作他们了.
4862                  */
4863    }
4864 });
4865
4866
4867
4868
4869
4870
4871
4872
4873
4874
4875
4876 //----------------------------------------------------------------------让每个jQuery 对象又具有了ajax 方面事件的监听能力------------------
4877 /*
4878  * 以下代码为"ajaxStart,ajaxStop,ajaxComplete,ajaxError,ajaxSuccess,ajaxSend"各定义一个事件绑定函数. 注意这些事件是自定义事件, 于是事件
4879  * 的触发就需要自己来了.
4880  */
4881 jQuery.each("ajaxStart,ajaxStop,ajaxComplete,ajaxError,ajaxSuccess,ajaxSend".split(","), function(i,o){
4882     jQuery.fn[o] = function(f){
4883         return this.bind(o, f);
4884     };
4885 });
4886 //----------------------------------------------------------------------
```

```
-----------------------------------------------------------
4887
4888
4889
4890
4891    //---------------------------------------------------------------------
        让jQuery 再具有ajax方面的静态函数 ----------------------------
4892    // 这些函数都是jQuery在完成ajax任务时所真正使用的底层静态函数.
        jQuery对象的ajax方法全部是对这些方法的封装. 这些静态函数构成了jQuery
        ajax的核心
4893    //---------------------------------------------------------------------
        ------------------------------------------------------------
4894    var jsc = now();//jsc被赋予当前的时间,作为一个时间戳.
4895
4896    jQuery.extend({
4897        /**
4898         * 使用GET方法发送请求
4899         * @param {string } url - 请求地址
4900         * @param {Object } data - 发送的数据
4901         * @param {Function} callback - 请求成功后需要执行的函数
4902         * @param {string} type - 请求的文档数据类型
4903         */
4904        get: function( url, data, callback, type ) {
4905            // shift arguments if data argument was ommited
4906            //
        如果data是一个函数而不是一个字符串那么就要做一个"矫正"的操作
4907            if ( jQuery.isFunction( data ) ) {
4908                callback = data;
4909                data = null;
4910            }
4911
4912            //调用jQuery.ajax函数
4913            return jQuery.ajax({
4914                type: "GET",//使用GET方法发送请求
4915                url: url,//路径
4916                data: data,//需要发送的数据
4917                success: callback,//成功响应后所要执行的函数
4918                dataType: type//期望响应的数据类型, 如application/xml,
        text/xml,text/html等
4919            });
4920        },
4921
4922        /**
4923         * 使用GET方法向url指定的地址请求一个脚本文件.
4924         * @param {string} url - 请求地址
4925         * @param {Function} callback - 请求成功后需要执行的函数
4926         */
4927        getScript: function( url, callback ) {
4928            return jQuery.get(url, null, callback, "script");
4929        },
4930        /**
4931         * 使用GET方法向url指定的地址请求JSON格式的数据
4932         * @param {string} 请求地址
4933         * @param {string} 发送的数据
4934         * @param {Function} 请求成功后需要执行的函数
4935         */
4936        getJSON: function( url, data, callback ) {
4937            return jQuery.get(url, data, callback, "json");
```

```
4938        },
4939
4940        /**
4941         * 使用POST方法向url指定的地址发送请求
4942         * @param {string } url - 请求地址
4943         * @param {Object } data - 发送的数据
4944         * @param {Function} callback - 请求成功后需要执行的函数
4945         * @param {string} type - 请求的文档数据类型
4946         */
4947        post: function( url, data, callback, type ) {
4948            //看看data是不是函数,如果是函数那就做一些"矫正"的工作.
    有的人喜欢这种简便的调用方式.
4949            if ( jQuery.isFunction( data ) ) {
4950                callback = data;
4951                data = {};
4952            }
4953
4954            //调用核心的ajax方法, 将发送方式设置为POST
4955            return jQuery.ajax({
4956                type: "POST",
4957                url: url,
4958                data: data,
4959                success: callback,
4960                dataType: type
4961            });
4962        },
4963        /**
4964         * 添加或者修改ajax默认参数设置.
4965         * 可以看到使用了extend函数来达到这个目的.
4966         *
    extend函数会看看jQuery.ajaxSettings有没有settings里面所列出的属性,没有
    就给它加上,有就用settings里的属性值代替jQuery.ajaxSettings
4967         * 里的同名属性的值.
4968         * @param {Object} settings
4969         */
4970        ajaxSetup: function( settings ) {
4971            jQuery.extend( jQuery.ajaxSettings, settings );
4972        },
4973
4974        /**
4975         * ajax的默认设置
4976         */
4977        ajaxSettings: {
4978            url: location.href,
4979            global: true,// 设置本次请求的作用范围是否为全局.
    像ajaxStart, ajaxStop, click, blur, focus 等事件都是global全局事件.
4980                         // 比如说我们的页面上有一个id为'panel'的div.
    这个div在ajax请求发送时显示"request sending...",结束后显示
4981                         // "stop". 我们可能会写如下的代码:
4982                         /*
4983                          * $('#panel').ajaxStart(function(){//set text
    'request sending...'}).ajaxStop(function(){// set text 'stop'});
4984                          *
4985                          *
4986                          * 如果我们的global参数为false,
    那么这些事件发生时并不触发绑定在这些事件上的监听函数,
    而只是运行用户在参数设置时所设定的
4987                          * callback. 在默认情况之下global为true,
```

```
也就是说我们在这些事件上绑定的事件监听函数都会得到运行.
4988                        */
4989        type: "GET",
4990        timeout: 0,
4991        contentType: "application/x-www-form-urlencoded",
4992        processData: true,
4993        async: true,
4994        data: null,
4995        username: null,
4996        password: null,
4997        accepts: {//请求所期望的(浏览能够接收的)数据类型,即MIME type
4998            xml: "application/xml, text/xml",
4999            html: "text/html",
5000            script: "text/javascript, application/javascript",
5001            json: "application/json, text/javascript",
5002            text: "text/plain",
5003            _default: "*/*"//默认情况下为所有的数据类型都能接收.
5004        }
5005    },
5006
5007    // Last-Modified header cache for next request
5008    // 翻译:为下一次请求缓存的Last-Modified头部.
5009    lastModified: {},
5010
5011    /**
5012     * jQuery ajax的核心方法. 用于根据设置发送ajax请求.
5013     * @param {Object} s - 发送设置.
5014     */
5015    ajax: function( s ) {
5016        // Extend the settings, but re-extend 's' so that it can be
5017        // checked again later (in the test suite, specifically)
5018        s = jQuery.extend(true, s, jQuery.extend(true, {}, jQuery.
ajaxSettings, s));
5019
5020        var jsonp, /*
JSONP是一个非官方的协议,它允许服务器端集成Script Tags返回给客户端,
通过JavaScript callback的形式简单实现跨域访问.
5021                    * 这是一个简单的jQuery JSONP
url的例子:"http://www.linhuihua.com?info=latestNews&callback=?&date=20
09-5-15", jQuery
5022                    * 使用下面那个正则表达式jsre将"=?"找出来,
然后替换成一个你指定的函数名称. 假设你所请求的响应数据为[{2009_5_15,
'No news today'}],
5023                    * 而你指定的callback名称为 displayNews,
则服务器返回 "<script>displayNews([{2009_5_15, 'No news
today'}])</script>".
5024                    *
5025                    *
更多关于JSONP的信息请参考wikipedia:http://en.wikipedia.org/wiki/JSONP#
JSONP. 如果你对JSONP没有兴趣,你仅需知道
5026                    * 认为这是一种实现JavaScript跨域访问的方式即可.
5027                    */
5028            jsre = /=\?(&|$)/g, // 解释看上面的中文注释
5029            status, //此属性是用来判断请求/响应状态的.
5030            data, //需要发送的数据.
5031            type = s.type.toUpperCase();//HTTP请求发送类型,GET or POST
5032
5033        // convert data if not already a string
```

```
5034            //
如果数据还没有被转化成字符串,那就调用param把他们转化为字符串.
5035            if ( s.data && s.processData && typeof s.data != "string" )
5036                s.data = jQuery.param(s.data);
5037
5038            // Handle JSONP Parameter Callbacks
5039            // 如果请求的数据类型是JSON,
那么就要根据请求中的url中是否含有"=?"来判断这是不是一个 jQuery
JSONP的请求, 然后做出相应的调整.
5040            if ( s.dataType == "jsonp" ) {
5041                if ( type == "GET" ) {
5042                    if ( !s.url.match(jsre) )//如果url不是一个jQuery
JSONP格式的url,那就把这个url修改成一个带有类似"callback=?"的url
5043                        s.url += (s.url.match(/\?/) ? "&" : "?") + (s.
jsonp || "callback") + "=?";
5044                } else if ( !s.data || !s.data.match(jsre) )
//没有要发送的data,或者有,不过data里面没有jsre所描述的那中url模式,都把
这些url组装成为
5045                                                        //jQuery
JSONP的格式.
5046                    s.data = (s.data ? s.data + "&" : "") + (s.jsonp ||
"callback") + "=?";
5047                s.dataType = "json";
//把dataType改会json,因为jsonp不是标准来的.
与此同时,将s.dataType修改成"json"之后, 代码才能进入下面那个
5048                                    //if语句来执行.
5049            }
5050
5051            // Build temporary JSONP function
5052            // 翻译: 建立临时的 JSONP 函数
5053            // 如果所请求的数据类型为json, 并且请求参数符合jQuery JSONP
的模式
5054            if ( s.dataType == "json" && (s.data && s.data.match(jsre) ||
 s.url.match(jsre)) ) {
5055                jsonp = "jsonp" + jsc++;//
jsc为当前时间的毫秒数,现在++了.现在jsonp将被用作jQuery自定义的JSONP的
回调函数的名称.例如,假设那个毫秒
5056                                        //
数为145386355672(这个数字我随便安的别跟我较真啊),那么最终生成的JSONP所
用的url长得可能是这个样子:
5057                                        //
http://www.linhuihua.com?show=newInfo&callback=jsonp145386355672&date=
2009-5-15
5058
5059                // Replace the =? sequence both in the query string and
the data
5060                // 将"=?"替换为jQuery在jsonp变量里面定义的callback名称,
在url地址, 发送的参数上都进行这个替换.
5061                if ( s.data )
5062                    s.data = (s.data + "").replace(jsre, "=" + jsonp +
"$1");
5063                s.url = s.url.replace(jsre, "=" + jsonp + "$1");
5064
5065                // We need to make sure
5066                // that a JSONP style response is executed properly
5067                s.dataType = "script";//将dataType最终改为"script",
这样浏览器在接收到JSONP响应(那不过是一些JS代码)之后能够运行加载回来的J
S代码.
```

```
5068
5069            // Handle JSONP-style loading
5070            /*
5071             * 上面完成了JSONP请求的构造,
而我们也在url中指定了callback的名称,
在上面的注释的例子中这个名称为jsonp145386355672.而下面的代码就
5072             * 是真正定义一个jsonp145386355672函数.
5073             */
5074            window[ jsonp ] = function(tmp){
//tmp是服务器的返回结果中的数据部分.
5075                data = tmp;
5076                success();//触发success事件
5077                complete();//触发complete事件
5078                // Garbage collect 垃圾回收
5079                window[ jsonp ] = undefined;
5080                try{ delete window[ jsonp ]; } catch(e){}
//运行完这个callback函数之后, 删除他的引用,
然后等待垃圾回收器将其回收.
5081                if ( head )// head 文档中<head>的引用
5082                    head.removeChild( script );
//将服务器返回的JS代码插入到浏览器, 浏览器就会运行这些代码.
5083            };
5084        }
5085
5086        //如果请求的响应数据是script,
并且在ajax参数设置上没有指定cache属性, 那就让cache为false,
不用缓存这些JS数据
5087        if ( s.dataType == "script" && s.cache == null )
5088            s.cache = false;
5089
5090        if ( s.cache === false && type == "GET" ) {
5091            var ts = now();//事件戳
5092            // try replacing _= if it is there
5093            // url中用"_="后边接一个日期或者毫秒数来表示时间戳,
如果ur中含有"_=",那证明这里有一个时间戳,那么修改这个事件戳的值为ts的值
5094            var ret = s.url.replace(/(\?|&)_=.*?(&|$)/, "$1_=" + ts +
 "$2");
5095            // if nothing was replaced, add timestamp to the end
5096            //
如果在上面的replace中并没有发生任何的改变,那说明url里没有时间戳,
那把ts这个时间戳放在url的后面.
5097            s.url = ret + ((ret == s.url) ? (s.url.match(/\?/) ? "&"
: "?") + "_=" + ts : "");
5098        }
5099
5100        // If data is available, append data to url for get requests
5101        // 翻译:如果有提供传送的数据,
把数据添加到请求url的后边(译者注:当然这个请求必须是GET请求)
5102        if ( s.data && type == "GET" ) {
5103            s.url += (s.url.match(/\?/) ? "&" : "?") + s.data;
5104
5105            // IE likes to send both get and post data, prevent this
5106            // COMP: 翻译:IE喜欢一起发送get和post的数据,
(采取措施)阻止它这样
5107            s.data = null;
5108        }
5109
5110        // Watch for a new set of requests
```

```
5111        if ( s.global && ! jQuery.active++ )
//jQuery.active为0时,才有可能运行下面的trigger代码.也就是说一个新的请求队列产生时, ajaxStart
5112            jQuery.event.trigger( "ajaxStart" );//触发ajaxStart时间, fire!
5113
5114        // Matches an absolute URL, and saves the domain
5115        // 翻译: 匹配一个绝对的网络地址(URL), 并保存它的域名
5116        var remote = /^(?:\w+:)?\/\/([^\/?#]+)/;
5117
5118        // If we're requesting a remote document
5119        // and trying to load JSON or Script with a GET
5120        //
对比本地域名与请求域名,如果不相同,则说明是一个远程请求,那就使用在<head>中动态添加<script>标签的做法来实现跨域请求
5121        if ( s.dataType == "script" && type == "GET"
5122                && remote.test(s.url) && remote.exec(s.url)[1] != location.host ){
5123            var head = document.getElementsByTagName("head")[0];
//获取<head>标签的引用
5124            var script = document.createElement("script");
//创建一个script标签
5125            script.src = s.url;//设置这个script节点的src属性,
当新建的script节点被插入<head>后, 浏览器将会根据这个地址加载脚本
5126            if (s.scriptCharset)//如果指定了脚本的编码方式,
那就设置编码方式
5127                script.charset = s.scriptCharset;
5128
5129            // Handle Script loading
5130            if ( !jsonp ) {// jsonp要有非undefined的值,
必须要具备几个条件:(1)s.dataType必须是json/jsonp;(2)url必须符合jQuery JSONP
5131                                //
的要格式要求.代码如果运行进这个if语句块,
说明ajax函数调用者无意进行JSONP调用(设置了非json/jsonp的dataType或url
5132                                // 并不符合jQuery JSONP格式的要求),
那好, 把刚才
5133                var done = false;
5134
5135                // Attach handlers for all browsers
5136                script.onload = script.onreadystatechange = function(){
5137                    if ( !done && (!this.readyState ||
5138                            this.readyState == "loaded" || this.readyState == "complete") ) {
5139                        done = true;
//将done设置为true,onreadystatechange如果再被调用就再也不会触发下面的success,complete事件了
5140                        success();//触发ajax 请求 success事件
5141                        complete();//触发ajax 请求complete事件
5142                        head.removeChild( script );
//把script标签移除, 因为它是为了实现跨域请求临时的生成的.
5143                    }
5144                };
5145            }
5146
5147            head.appendChild(script);
//将新建的script节点加入到<head>标签中, 那浏览器就会加载这个脚本
```

```
5148
5149            // We handle everything using the script element injection
5150            //
注意整个if语句的条件:"如果使用GET方式请求一个远程script脚本".
那么,代码运行到这里事情已经完成了,接下来的事情全部由加载回来的JS代码完成.
5151            // 可以返回了. 返回值设置为undeined.
5152            return undefined;
5153        }
5154
5155
5156        /*
5157         * 在浏览器中有三种方式能够发送异步的HTTP请求:
5158         * (1) iframe
5159         * (2) script
5160         * (3) XMLHTTPRequest
5161         * 其中iframe由于安全性的问题, 一直遭人诟病.
5162         *
以本块注释为界,上面的代码使用了script的方式来发送一个请求脚本的异步请求;
5163         * 而本注释块下方的代码, 则使用了XMLHTTPRequest来发送请求.
5164         */
5165
5166        var requestDone = false;
5167
5168        // Create the request object; Microsoft failed to properly
5169        // implement the XMLHttpRequest in IE7, so we use the
ActiveXObject when it is available
5170        /* 翻译: 创建请求对象;
微软在IE7上并没有正确地实现XMLHttpRequest,
所以(为了安全起见)在ActiveXObject可用的时候尽量使用ActiveXObject
5171         * (来创建XMLHttpRequest).
5172         *
5173         * 出于以上(翻译)原因,
jQuery创建一个XMLHTTPRequest(下成XHR)对象的方式是比较简单的:对IE统一采用ActiveXObject, 而其他浏览器则使用
5174         *
XMLRequest().其实我们随便google出来的创建XHR代码都比jQuery的复杂,
"Microsoft.XMLHTTP"这个ActiveXObject是为了向后兼容最老版
5175         * 本的IE(IE5)上的XHR.
创建XHR的ActiveXObject其实还有更加新的版本:
5176         * (1) Msxml2.XMLHTTP.6.0
5177         * (2) Msxml2.XMLHTTP.3.0
5178         * (3) Msxml2.XMLHTTP
5179         * (4) Microsoft.XMLHTTP
5180         *
5181         * 在jQuery 1.3.2中,
下面的这行代码被移入了ajaxSettings.xhr函数中,并且能够被重写.
这样,在jQuery 1.3.2中如果你觉得jQuery的xhr创建
5182         * 方法太过简陋, 或者说你想对XHR对象进行缓存,
你可以自己调用ajaxSetup方法来自己定制.
不过,在这一个版本(1.2.6)的jQuery中, 恐怕我们就只
5183         * 能睁一只眼闭一只眼了.
5184         */
//IE
//non-IE(XHR正处于标准化的过程中,但目前还不是标准)
5185        var xhr = window.ActiveXObject ? new ActiveXObject(
"Microsoft.XMLHTTP") : new XMLHttpRequest();
```

```
5186
5187            // Open the socket
5188            // 翻译:打开socket
5189            /* 对于Resig师父所说的"socket", 《Pro JavaScript Techniques
    (精通JavaScript)》的译者有话要说:
5190             * "...open the socket(打开套接字), 这是错误的.
    考虑XHR对象完全不需要考虑到socket编程的细节, Microsoft, Apple 和
    Mozilla的文档
5191             * 也无一提到过open和socket有任何关系,
    都只是说open用于初始化一个准备发起仍在'pending'状态中的请求..."
5192             *
5193             * so, 我只是列出一些不同的声音, 请聪明的读者自行甄别.
5194             */
5195            //
5196            // Passing null username, generates a login popup on Opera
    (#2865)
5197            // comp: 在Opera 9.5(至少在这一个版本)中,
    如果我们使用XHR来发送一个请求, 并且s.username是一个null值时,
    浏览器会弹出一个prompt框要求
5198            // 用户登录. 于是为了避免这个问题,
    我们对username进行区别对待, 有username, 就把他发过去; 没有就不要发,
    而不是发一个null到服务器端.
5199            if( s.username )
5200                xhr.open(type, s.url, s.async, s.username, s.password);
5201            else
5202                xhr.open(type, s.url, s.async);
5203
5204            // Need an extra try/catch for cross domain requests in
    Firefox 3
5205            //
    翻译:在Firefox3中发送跨域请求需要一个额外的try/catch块(译者注:Firefox3
    中这些设置能会引发错误,故需要使用一个try/catch)
5206            try {
5207                // Set the correct header, if data is being sent
5208                // 如果有要发送的data的话, 那要好好设置这些data的类型
5209                if ( s.data )
5210                    xhr.setRequestHeader("Content-Type", s.contentType);
5211
5212                // Set the If-Modified-Since header, if ifModified mode.
5213                // 如果需要一个过期头, 那就设置这个过期头.
    过期头所标识的日期一般用于浏览器的缓存设置.
    如果服务器端那边的页面的更新日期要晚于过期头内
5214                // 标注的日期, 服务器端就返回这个最近更新的页面;
    否则返回304状态: not-modified, 浏览器就不用加载一样的页面,
    提升了用户体验.
5215                if ( s.ifModified )
5216                    xhr.setRequestHeader("If-Modified-Since",
5217                        jQuery.lastModified[s.url] || "Thu, 01 Jan 1970
    00:00:00 GMT" );
5218
5219                // Set header so the called script knows that it's an
    XMLHttpRequest
5220                // 翻译:设置(相应的)头部,
    这样(服务器端)脚本便能知道这是一个通过XMLHttpRequest发送的请求.
5221                xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest"
    );
5222
5223                // Set the Accepts header for the server, depending on
```

```
the dataType
5224                // 设置接收数据的类型,
好让服务器知道该给你返回什么类型的数据; 置于具体是什么类型,
这个由dataType参数来决定.
5225                xhr.setRequestHeader("Accept", s.dataType && s.accepts[ s
.dataType ] ?
5226                    s.accepts[ s.dataType ] + ", */*" :
5227                    s.accepts._default );//
5228            } catch(e){}
5229
5230            // Allow custom headers/mimetypes
5231            // beforeSend是用户自己在传入进来的参数中定义的一个方法,
这样用户就可以在XHR请求发送之前做一些自己想做的事情, 干预请求
5232            if ( s.beforeSend && s.beforeSend(xhr, s) === false /*
如果beforedSend返回false则取消XHR请求 */ ) {
5233                // cleanup active request counter
5234                // XHR请求被取消, 活跃的XHR请求数当然要减1啦.
5235                s.global && jQuery.active--;
5236                // close opended socket
5237                // 翻译:关闭已经打开的请求
5238                xhr.abort();
5239                return false;// 返回false说明ajax请求发送失败.
5240            }
5241
5242            //global默认是true
5243            if ( s.global )
5244                jQuery.event.trigger("ajaxSend", [xhr, s]);// 好吧,
一切准备就绪, 触发发送事件, 绑定在这个事件上的事件监听函数将会被运行
5245
5246
5247
5248            /*
5249             * XMLHTTPRequest方式的异步请求发送部分设置完毕,
下面进行响应接收部分的设置:
5250             */
5251
5252            // Wait for a response to come back
5253            // 翻译: 等待返回的请求
5254            var onreadystatechange = function(isTimeout){
5255                // The transfer is complete and the data is available,
or the request timed out
5256                // 翻译: 传输完毕 并且数据可用,
或者请求超时我们都认为本次请求完成(requestDone = true)
5257                if ( !requestDone && xhr && (xhr.readyState == 4 ||
isTimeout == "timeout") ) {
5258                    requestDone = true;
5259
5260                    // clear poll interval
5261                    // ival是计时器的引用, 如果有这个引用,
证明设置了计时器, 请求完成就当然要清除这个计时器啦,
这样,请求就不会重试了.
5262                    if (ival) {
5263                        clearInterval(ival);//清除请求
5264                        ival = null;
5265                    }
5266
5267                    //获取请求完成后的请求状态:
5268                    // 注意 "status = isTimeout == "timeout" &&
```

```
"timeout" ||" 这句代码的执行顺序是
5269                // "status = (isTimeout == "timeout") && "timeout"
5270                status = isTimeout == "timeout" && "timeout" ||
5271                    !jQuery.httpSuccess( xhr ) && "error" ||
5272                    s.ifModified && jQuery.httpNotModified( xhr, s.
url ) && "notmodified" ||
5273                    "success";
5274
5275
//如果请求成功就调用jQuery.httpData函数来解析请求回来的数据.解析的过程
中如果出错, 就设置status为:"parsererror"
5276                if ( status == "success" ) {
5277                    // Watch for, and catch, XML document parse errors
5278                    try {
5279                        // process the data (runs the xml through
httpData regardless of callback)
5280                        data = jQuery.httpData( xhr, s.dataType, s.
dataFilter );
5281                    } catch(e) {
5282                        status = "parsererror";
5283                    }
5284                }
5285
5286                // Make sure that the request was successful or
notmodified
5287                if ( status == "success" ) {
5288                    // Cache Last-Modified header, if ifModified mode.
5289                    // 如果设置了ifModified为true,
说明要对响应头进行缓存(这样下次请求相同url的时候可以看看请求的页面的修
改日期是否晚过这个日期,
5290                    //
从而决定是否加载那个页面).下面代码的主要工作就是要保存这个last-Modifie
d
5291                    var modRes;
5292                    try {
5293                        modRes = xhr.getResponseHeader(
"Last-Modified");
5294                    } catch(e) {} // swallow exception thrown by FF
if header is not available
5295
5296                    if ( s.ifModified && modRes )
5297                        jQuery.lastModified[s.url] = modRes;
//保存这个last-Modified的时间
5298
5299
5300
5301                    // JSONP handles its own success callback
5302                    // JSONP 有自己的success callback,
不需要运行下面这个success函数.
5303                    if ( !jsonp )
5304                        success();
5305                } else//如果不是"success"那就认为是出错了,
调用jQuery.handleError函数来处理这个情况.
5306                    jQuery.handleError(s, xhr, status);
5307
5308                // Fire the complete handlers
5309                // 翻译: 触发complete事件,
那么绑定在这个事件上事件监听函数就会被运行.
```

```
5310                    complete();
5311
5312                    // Stop memory leaks
5313                    // 把xhr设为null, 让垃圾回收器对xhr进行回收,
    防止内存泄漏.
5314                    if ( s.async )//s.async在默认的情况之下是true,
    使用异步的方式发送请求.
5315                        xhr = null;
5316                }
5317            };
5318
5319            //如果是异步的请求, 设置请求重试, 一次不成功就再来一次,
    直到成功或者超时
5320            if ( s.async ) {
5321                // don't attach the handler to the request, just poll it
    instead
5322                var ival = setInterval(onreadystatechange, 13);
5323
5324                // Timeout checker
5325                // 设置超时后的处理函数和超时的时间.
5326                if ( s.timeout > 0 )
5327                    setTimeout(function(){
5328                        // Check to see if the request is still happening
5329                        // 如果xhr不为null, 说明请求正在进行,
    取消这次请求, 因为超时了
5330                        if ( xhr ) {
5331                            // Cancel the request
5332                            xhr.abort();
5333
5334                            if( !requestDone )//如果请求还没完成,
    不管了,
    马上调用onreadystatechange并传入"timeout",这样requestDone就会==true
5335                                onreadystatechange( "timeout" );
5336                        }
5337                    }, s.timeout);
5338            }
5339
5340            /*
5341             * 好了,好了...发送的设置, 响应的设置总算完成了, 可以发送了.
5342             */
5343
5344            // Send the data
5345            // 终于可以发送请求了
5346            try {
5347                xhr.send(s.data);
5348            } catch(e) {
5349                //出错就调用handleError进行处理
5350                jQuery.handleError(s, xhr, null, e);
5351            }
5352
5353            // firefox 1.5 doesn't fire statechange for sync requests
5354            // COMP:翻译:在firefox 1.5中,
    同步请求并不能触发statechange事件. (好吧, 自己来... )
5355            if ( !s.async )
5356                onreadystatechange();//自己触发这个事件
5357
5358            /**
5359             * 请求成功事件的触发函数
```

```
5360            */
5361           function success(){
5362               // If a local callback was specified, fire it and pass
    it the data
5363               // 翻译:如果用户提供了自己的callback函数,
    就在这里调用它, 并把数据传给它
5364               if ( s.success )
5365                   s.success( data, status );
5366
5367               // Fire the global callback
5368               // 翻译:触发全局的callback
5369               //
    如果有其他的元素的处理事件绑定到了这个事件(ajaxSuccess)上,
    触发这些函数
5370               if ( s.global )
5371                   jQuery.event.trigger( "ajaxSuccess", [xhr, s] );
5372           }
5373           /**
5374            * 请求请求发送完成事件的触发函数
5375            */
5376           function complete(){
5377               // Process result
5378               // 如果用户提供了自己的complete callback函数,
    就在这里调用它, 并把数据传给它
5379               if ( s.complete )
5380                   s.complete(xhr, status);
5381
5382               // The request was completed
5383               // 触发全局的callback, 触发其他绑定到这个事件上的函数.
5384               if ( s.global )
5385                   jQuery.event.trigger( "ajaxComplete", [xhr, s] );
5386
5387               // Handle the global AJAX counter
5388               // 如果全局的活跃请求数目为0, 触发ajaxStop事件,
    绑定在其上的事件监听函数得到运行.
5389               if ( s.global && ! --jQuery.active )
5390                   jQuery.event.trigger( "ajaxStop" );
5391           }
5392
5393           // return XMLHttpRequest to allow aborting the request etc.
5394           // 意译: 返回xhr, 这样做的作用有很多,
    比如说可以随时取消这个请求等.
5395           return xhr;
5396       },
5397
5398       /**
5399        * jQuery.ajax方法中出现的错误处理函数
5400        * @param {Object} s - ajax设置
5401        * @param {XMLHTTPRequest} xhr
5402        * @param {string} status - ajax请求状态, 如success, timout等
5403        * @param {Object} e - 错误出现时, JavaScript解析器抛出的错误对象
5404        */
5405       handleError: function( s, xhr, status, e ) {
5406           // If a local callback was specified, fire it
5407           // 如果用户有提供错误发生时可以调用的回调函数, 那就调用它咯
5408           if ( s.error ) s.error( xhr, status, e );
5409
5410           // Fire the global callback
```

```
5411            // 如果ajax请求是全局的, 触发ajaxError事件.
5412            if ( s.global )
5413                jQuery.event.trigger( "ajaxError", [xhr, s, e] );
5414        },
5415
5416        // Counter for holding the number of active queries
5417        active: 0,//活跃的请求数, 以此来计数到底有多少个请求等待发送出去.
5418
5419        // Determines if an XMLHttpRequest was successful or not
5420        /**
5421         * 翻译: 判断当前这个请求是否是成功的.
5422         * @param {XMLHTTPRequest} xhr
5423         */
5424        httpSuccess: function( xhr ) {
5425            try {
5426                // IE error sometimes returns 1223 when it should be 204
    so treat it as success, see #1450
5427                // IE有一个错误, 那就是有时候应该返回204(No
    Content)但是它却返回1223, 好吧, 把这种情况也算作是请求成功
5428                // 详细请看链接:http://dev.jquery.com/ticket/1450,
    似乎也没有很好地解决这个问题.
5429                return !xhr.status && location.protocol == "file:" ||
    //如果本地文件的, 没有status也是成功的请求,这种情况返回true;
5430                    ( xhr.status >= 200 && xhr.status < 300 ) || xhr.
    status == 304 || xhr.status == 1223 ||//这里列出了可以认为是成功的
5431
    //safari在文档没有修改时(304)得到的status会等于undefined,
    所以把这种情况也当作是成功
                                                    //请求的状态码.
5432                    jQuery.browser.safari && xhr.status == undefined;
5433            } catch(e){}
5434            return false;//代码还能运行到这样里, 证明真的是失败了.
5435        },
5436
5437        // Determines if an XMLHttpRequest returns NotModified
5438        /**
5439         * 判断请求回来的服务器响应是不是"NotModified".
5440         * @param {XMLHTTPRequest} xhr
5441         * @param {string} url
5442         */
5443        httpNotModified: function( xhr, url ) {
5444            try {
5445                var xhrRes = xhr.getResponseHeader("Last-Modified");
5446
5447                // Firefox always returns 200. check Last-Modified date
5448                // 翻译: Firefox 总是返回200.
    还是对比一下Last-Modified的日期稳妥一些.
5449                return xhr.status == 304 || xhrRes == jQuery.lastModified
    [url] ||
5450                    jQuery.browser.safari && xhr.status == undefined;//
    safari在文档没有修改时(304)得到的status会等于undefined
5451            } catch(e){}
5452            return false;//代码还能运行到这样里,
    证明真的不是"NotModified".
5453        },
5454        /**
5455         *
    获取XMLHTTPRequest的响应数据.允许对数据使用自定义的函数进行预处理.并根
```

```
据用户提供的数据类型对响应数据做不同的处理. 最后将数据返回.
5456        * @param {XMLHTTPRequest} xhr
5457        * @param {String} type - 响应的数据类型名称,如json,xml等.
不是MIME type
5458        * @param {Function} filter - 预处理函数
5459        */
5460       httpData: function( xhr, type, filter ) {
5461           var ct = xhr.getResponseHeader("content-type"),
5462               xml = type == "xml" || !type && ct && ct.indexOf("xml")
>= 0,
5463               data = xml ? xhr.responseXML : xhr.responseText;//
如果响应不是XML就统一把数组当作普通文本.等下再根据用户提供的数据类型,
将文
5464                                                                //
本转化成为相应的数据类型.
5465
5466           //这个条件比较搞笑: 当出现请求响应错误的时候,
有服务器会返回一个XML文档来描述这个错误.
那么在这种情况之下我们当然不能认为这个请求成功啦,抛出一个错误
5467           if ( xml && data.documentElement.tagName == "parsererror" )
5468               throw "parsererror";
5469
5470           // Allow a pre-filtering function to sanitize the response
5471           // 翻译: 允许一个预过滤函数对响应数据进行"消毒"...
5472           if( filter )//如果有提供一个过滤函数,
那就调用这个过滤函数首先对响应的数据进行预处理
5473               data = filter( data, type );
5474
5475           // If the type is "script", eval it in global context
5476           // 翻译: 如果类型是script, 那么在全局的上下文环境中运行它
5477           if ( type == "script" )
5478               jQuery.globalEval( data );
5479
5480           // Get the JavaScript object, if JSON is used.
5481           // 翻译: 如果请求的是json数据, 用eval获取JavaScript object
5482           if ( type == "json" )
5483               data = eval("(" + data + ")");
5484
5485           return data;//把获得的数据返回
5486       },
5487
5488       // Serialize an array of form elements or a set of
5489       // key/values into a query string
5490       //
5491       //
5492       /**
5493        * 串行化一个装着表单元素的数组 或者 是一个键值对的集合.
这里是一个串行化的例子:{'name':'auscar','university':'SYSU'}串行化之后
变为一个字
5494        * 符串:"name=auscar&university:SYSU".
5495        * @param {Object} a - 需要串行化的对象
5496        */
5497       param: function( a ) {
5498           var s = [];//最终结果集
5499
5500           // If an array was passed in, assume that it is an array
5501           // of form elements
5502           // 翻译: 如果一个数组传进来了,
```

```
        那就假设它是一个有表单元素组成的数组.
5503            if ( a.constructor == Array || a.jquery )
        //数组或者类数组(jQuery对象就是一个类数组)都需要遍历:
5504                // Serialize the form elements 翻译: 串行化标点元素
5505                jQuery.each( a, function(){
5506                    //对键名和键值进行编码后就存进了最终结果集,
        最后返回前会用"&"符号将他们连接起来.
5507                    s.push( encodeURIComponent(this.name) + "=" +
        encodeURIComponent( this.value ) );
5508                });
5509
5510            // Otherwise, assume that it's an object of key/value pairs
5511            // 如果不是数组, 那就假设这是一个由键/值对组成的对象.
5512            else
5513                // Serialize the key/values
5514                // 翻译: 串行化键/值
5515                for ( var j in a )
5516                    // If the value is an array then the key names need
        to be repeated
5517                    if ( a[j] && a[j].constructor == Array )//
        如果键/值里的值是一个数组, 那么就要再遍历这个数组,
        然后进行同上面数组一样的字符
5518                                                            // 串拼接.
5519                        jQuery.each( a[j], function(){
5520                            //
        从下面的代码可以看到类似{favourite:['JavaScript', 'tennis',
        'guitar']}的对象会被转化成:
5521                            //
        "favourite=JavaScript&favourite=tennis&favourite=guitar"
5522                            s.push( encodeURIComponent(j) + "=" +
        encodeURIComponent( this ) );
5523                        });
5524                    else

        //从这里可以看到, 键/值中的值还可以是一个Function
5525                        s.push( encodeURIComponent(j) + "=" +
        encodeURIComponent( jQuery.isFunction(a[j]) ? a[j]() : a[j] ) );
5526
5527            // Return the resulting serialization
5528            return s.join("&").replace(/%20/g, "+");
        //最后返回串行化后的字符串结果.
5529        }
5530
5531    });
5532    //-----------------------------------------------------------------
        ----------------------------------------------------------------
5533
5534
5535
5536
5537
5538
5539    //-----------------------------------------------------------------
        ----------给jQuery 对象添加基本动画方法-----------------------------
5540    jQuery.fn.extend({
5541        /**
5542         * 以speed所指示的速度显示jQuery对象匹配元素集合中的对应元素.
5543         * 如果函数被以无参的形式调用,
```

```
那么jQuery对象中的匹配元素集合中隐藏的元素就会被"一下子"显示出来,
没有渐变的过程.
5544         * @param {Object} speed - 显示的速度,
也就是说显示元素的过程持续多长时间.
可以是Number也可以是"slow","normal","default"三者之一.
5545         * @param {Function} callback
5546         */
5547        show: function(speed,callback){
5548
//如果提供了speed参数,则使用animate函数设置本次show动画的时间长度,然后
再进行动画
5549            return speed ?
5550                this.animate({
5551                    height: "show", width: "show", opacity: "show"
5552                }, speed, callback) :
5553
5554                //如果没有传入speed, 也就是说不带参调用show,
那就让所有的带有"hidden"样式的元素都show出来.
5555                this.filter(":hidden").each(function(){
5556                    this.style.display = this.oldblock || "";
5557
5558                    //经过上面一句的赋值之后,
如果this(this指向的是一个HTML元素)的oldblock为none,
那就赋予它一个非none的值. 那, 到底是
5559                    //什么值呢? 这需要经过一定处理才能确定,
下面就是这个确定的过程:
5560                    if ( jQuery.css(this,"display") == "none" ) {
5561
5562                        //好吧, 既然你是none,
那我就看看元素所属的标签默认是使用什么display值的.
于是新建一个与元素的标签名一样的临时元素,
5563                        //并把它插入到body里面.
5564                        var elem = jQuery("<" + this.tagName + " />").
appendTo("body");
5565
5566                        //
然后就看看这种元素的display在默认情况之下是什么属性值,
并把这个默认值赋予this.style.display
5567                        this.style.display = elem.css("display");
5568
5569                        // handle an edge condition where css is - div {
display:none; } or similar
5570                        // 哎呀, 如果还是none, 不行, 你至少要是block
5571                        if (this.style.display == "none")
5572                            this.style.display = "block";
5573                        elem.remove();//OK,
在获取元素的display默认属性值之后, 将这个临时的元素删除.
5574                    }
5575                })
5576                .end();//
因为filter函数对jQuery对象的匹配元素集合产生了"破坏性"的影响,即改变了j
Query对象的匹配元素集合的内容. 因此需要
5577                       //
使用jQuery.fn.end函数将匹配元素集合恢复到filter之前的状态.
5578        },
5579        /**
5580         * 顾名思义, 将jQuery对象匹配元素集合中的元素隐藏起来.
5581         * @param {Object} speed - 显示的速度,
```

```
也就是说显示元素的过程持续多长时间.
可以是Number也可以是"slow","normal","default"三者之一.
5582         * @param {Object} callback - 隐藏动作完成之后需要执行的函数.
5583         */
5584        hide: function(speed,callback){
5585            return speed ?
5586                // 如果传入了speed 参数, 那就按照speed的速度要求,
动画呈现隐藏的过程.
5587                this.animate({
5588                    height: "hide", width: "hide", opacity: "hide"
5589                }, speed, callback) :
5590
5591                this.filter(":visible").each(function(){
5592                    this.oldblock = this.oldblock || jQuery.css(this,
"display");// 保存元素当前的blcok属性, 当需要再次显示时将这
5593
            // 个属性设置回去.
5594                    this.style.display = "none";// 让元素隐藏起来,
没有任何的渐变效果.
5595                })
5596                .end();//
因为filter函数对jQuery对象的匹配元素集合产生了"破坏性"的影响,即改变了j
Query对象的匹配元素集合的内容. 因此需要
5597                        //
使用jQuery.fn.end函数将匹配元素集合恢复到filter之前的状态.
5598        },
5599
5600        // Save the old toggle function
5601        // 翻译: 保存旧的toggle函数.
5602        // 旧的toggle函数可以接收任意数量的函数作为参数.
旧toggle函数的作用就是在传入的函数之间轮流调用它们. 详细情况,
请参考jQuery.fn.toggle
5603        // 的中文注释.
5604        _toggle: jQuery.fn.toggle,
5605
5606        /**
5607         * 这个函数只接收两个参数: fn 和 fh2;
5608         *
toggle函数的作用就是在fn和fn2之间切换运行(如果fn和fn2都是函数的话).
5609         * 如果fn和fn2是两个对象,
那么就把fn当作是animate函数的speed参数,
把fn2当作是animate函数的easing参数, 然后调用animate函数进行动画.
5610         * 如果fn和fn2都为undefined, 即toggle函数被以无参的形式调用,
则让jQuery对象的匹配元素集合中元素在show和hide两种状态中切换.
5611         * @param {Object} fn
5612         * @param {Object} fn2
5613         */
5614        toggle: function( fn, fn2 ){
5615                //如果fn和fn2都是函数的话, 那调用旧的toggle方法,
在fn和fn2之间切换着调用.
5616            return jQuery.isFunction(fn) && jQuery.isFunction(fn2) ?
5617                this._toggle.apply( this, arguments ) :
5618
5619                // 如果两个都不是函数, 看看fn是否有值.
这主要是判断toggle函数是否是在以无参的形式调用
5620                fn ?
5621                    this.animate({//有参数的情况,
则把fn当作是animate函数的speed参数,
```

```
        把fn2当作是animate函数的easing参数, 然后调用animate函数进行动画
5622                    height: "toggle", width: "toggle", opacity:
    "toggle"
5623                }, fn, fn2) :
5624
5625                this.each(function(){//无参的情况,
    则让jQuery对象的匹配元素集合中元素在show和hide两种状态中切换.
5626                    jQuery(this)[ jQuery(this).is(":hidden") ? "show"
     : "hide" ]();
5627                });
5628    },
5629
5630    /**
5631     * 让匹配元素集合中的元素以一个"滑动着出来"的效果呈现
5632     * @param {Object} speed -
    三种预定速度的之一的字符串("slow","def","fast").
5633     * @param {Function} callback - 动画完成时所需要调用的函数.
5634     */
5635    slideDown: function(speed,callback){
5636        return this.animate({height: "show"}, speed, callback);
5637    },
5638
5639    /**
5640     * 让匹配元素集合中的元素以一个"滑动着"的效果消失
5641     * @param {Object} speed -
    三种预定速度的之一的字符串("slow","def","fast").
5642     * @param {Function} callback - 动画完成时所需要调用的函数.
5643     */
5644    slideUp: function(speed,callback){
5645        return this.animate({height: "hide"}, speed, callback);
5646    },
5647
5648    /**
5649     * 让匹配元素集合中的元素如果原本是slideUp的现在旧slideDown,
    原本是slideDown的, 现在是slideUp.
5650     * @param {Object} speed -
    三种预定速度的之一的字符串("slow","def","fast").
5651     * @param {Function} callback - 动画完成时所需要调用的函数.
5652     */
5653    slideToggle: function(speed, callback){
5654        return this.animate({height: "toggle"}, speed, callback);
5655    },
5656
5657    /**
5658     * 淡进
5659     * @param {Object} speed -
    三种预定速度的之一的字符串("slow","def","fast").
5660     * @param {Function} callback - 动画完成时所需要调用的函数.
5661     */
5662    fadeIn: function(speed, callback){
5663        return this.animate({opacity: "show"}, speed, callback);
5664    },
5665
5666    /**
5667     * 淡出
5668     * @param {Object} speed -
    三种预定速度的之一的字符串("slow","def","fast").
5669     * @param {Function} callback - 动画完成时所需要调用的函数.
```

```
5670         */
5671        fadeOut: function(speed, callback){
5672            return this.animate({opacity: "hide"}, speed, callback);
5673        },
5674
5675        /**
5676         * 让匹配元素集合中的元素的透明度以动画的形式显示到to所指定的值.
5677         * @param {Object} speed -
三种预定速度的之一的字符串("slow","def","fast").
5678         * @param {Object} to - opacity变化到to所指定的值.
5679         * @param {Function} callback - 动画完成时所需要调用的函数.
5680         */
5681        fadeTo: function(speed,to,callback){
5682            return this.animate({opacity: to}, speed, callback);
5683        },
5684
5685
5686        /**
5687         *  用于创建自定义动画的函数。
5688         *
这个函数的关键在于指定动画形式及结果样式属性对象(参数prop)。这个对象中
每个属性都表示一个可以变化的样式属性（如"height"、"top"或"opacity"）。
5689         *
注意：所有指定的属性必须用骆驼形式，比如用marginLeft代替margin-left.
5690         *
5691         *
而每个属性的值表示这个样式属性到多少时动画结束。如果是一个数值，样式属
性就会从当前的值渐变到指定的值。如果使用的是"hide"、"show"
5692         *  或"toggle"这样的字符串值，则会为该属性调用默认的动画形式。
5693         * @param {Object} prop - 一个对象,
它包含了当动画完成时动画对象所应当呈现的一组样式和这些样式的终值.
5694         * @param {string} speed -
三种预定速度的之一的字符串("slow","def","fast").
5695         * @param {Object} easing - 动画擦除效果,
目前jQuery只提供"liner"和"swing"两种效果.
不过jQuery也支持第三方的动画效果, 不过需要插件支持.
5696         * @param {Function} callback - 动画完成时所需要调用的函数.
5697         */
5698        animate: function( prop, speed, easing, callback ) {
5699            //将speed, easing,
callback三个参数使用speed函数组合到一个对象中,
这个对象由jQueury.speed返回. speed的细节请查看jQuery.speed的中文注释
5700            var optall = jQuery.speed(speed, easing, callback);
5701
5702
5703            // 执行 each 或者 queue函数
5704                    //这个this 指的是一个jQuery 对象
5705            return this[ optall.queue === false ? "each" : "queue" ](
function(){
5706                    //这个this 指的是一个普通的DOM元素
5707                if ( this.nodeType != 1)
5708                    return false;
5709
5710                var opt = jQuery.extend({}, optall), //复制optall
5711                                                    p,
5712                                                    hidden = jQuery(this).
is(":hidden"),
5713                                                    self = this;
```

```
//这个this 指的是一个普通的DOM元素
5714
5715
5716            // 对animate函数的第一个参数( 它是一个列表 )内的几个特殊的属性进行处理
5717            // 这些属性是:hide, show, height, width
5718            for ( p in prop ) {
5719                // 看看 hide属性,嗯,hidden了, 再看看show 属性,也show( !hidden) 了,看来已经完成了动画,可以执行complete了
5720                if ( prop[p] == "hide" && hidden || prop[p] == "show" && !hidden )
5721                    return opt.complete.call(this); //这个complete最后会执行传进来的callback, 不过在此之前它会进行一些额外操作
5722
5723                // 如果要进行动画的属性是height或者是width
5724                if ( p == "height" || p == "width" ) {
5725                    // Store display property
5726                    opt.display = jQuery.css(this, "display");
5727
5728                    // Make sure that nothing sneaks out
5729                    opt.overflow = this.style.overflow;
5730                }
5731            }
5732
5733            if ( opt.overflow != null )
5734                this.style.overflow = "hidden";
5735
5736            //curAnim 装的是 用户设置那些要进行动画的属性
5737            opt.curAnim = jQuery.extend({}, prop);
5738
5739            // 开始对prop 里面设置的每一个属性进行动画了
5740            jQuery.each( prop, function(name, val){
5741
5742                // 新建一个 fx 动画对象, name 是prop 里面 键/值 的 " 键 " ( 双引号表示强调... )
5743                // 注意,第一参数self是一个HTML元素的引用
5744                //     第二个参数是属性动画属性的设置集合,里面有类似"complete","duration","easing"之类的属性.还有一个属性的集合curAnim,其内装着用户设置的属性值
5745                //
第三个参数就是用户设置的属性集合中的一个属性的名称, 表示接下来要为这个属性新建一个fx动画对象
5746                //
这里是为每一个要进行动画的属性都新建了一个fx对象
5747                var e = new jQuery.fx( self, opt, name ); //为当前name所指定的属性新建一个动画函数.
5748
5749                // 看看可不可以调用基本的动画函数
5750                if ( /toggle|show|hide/.test(val) )
5751                    e[ val == "toggle" ? hidden ? "show" : "hide" : val ]( prop );//如果可以就直接调用这些基本的动画函数.
5752
5753                // 看来不需要调用基本动画函数
5754                else {
5755                    // 看看属性值是不是类似 " left:+50px " ,在 jQuery 1.2 中, 你可以通过在属性值前面指定 "+=" 或 "-="
来让元素做相对运动
```

```
5756                        var parts = val.toString().match(
/^([+-]=)?([\d+-.]+)(.*)$/),
5757                            start = e.cur(true) || 0;
//使用jQuery.fx对象的cur获取当前元素的样式. 注意cur的参数为true,
这表示直接获取
5758                                                    //元素的内联样式.
5759
5760                        // 如果parts
不是null,说明要做动画,注意match返回值不是boolean 类型, 而是一个数组
.数组里面装的是match到的字符串,下标由
5761                        // 正则表达式内的分组号决定
5762                        if ( parts ) {
5763                            var end = parseFloat(parts[2]),
//样式动画的终值
5764                                unit = parts[3] || "px";
//如果没有提供单位就使用"px"作为默认单位
5765
5766                            // We need to compute starting value
5767                            // 翻译: 我们需要计算出开始值
5768                            if ( unit != "px" ) {
5769                                self.style[ name ] = (end || 1) + unit;
5770                                start = ((end || 1) / e.cur(true)) *
start;
5771                                self.style[ name ] = start + unit;
5772                            }
5773
5774                            // If a +=/-= token was provided, we're
doing a relative animation
5775                            // 如果val中含有"+=/-=", 那么就做一次相对动画.
5776                            if ( parts[1] )
5777                                end = ((parts[1] == "-=" ? -1 : 1) * end)
 + start;//计算出最终的样式值.
5778
5779                            e.custom( start, end, unit );
//调用jQuery.fx对象的custom方法进行一个动画.
5780                        }
5781                        // 不是 left:+=5px这种类型,而是left:5px这种类型
5782                        else
5783                            e.custom( start, val, "" );
5784                    }
5785                });
5786
5787                // For JS strict compliance
5788                return true;
5789            });
5790        },
5791        /**
5792         * 功能有2:
5793         * (1) 获取匹配元素集合中首元素的动画队列;
5794         * (2) 设置匹配元素集合中每一个元素的动画队列.
可能是整个队列的替换, 又可能是仅仅将某个函数加入到队列的尾部.
5795         * @param {string} type - 动画的类型. 一般是"fx".
在拥有其他的扩展动画库的情况下, 这个值将会有变.
5796         * @param {Function or Array} fn - 如果是Function,
那就把Function加入到每个匹配元素的动画队列的尾部; 如果是Array那就用fn
5797         *                                      替换每个匹配元素的动画队列.
5798         */
5799        queue: function(type, fn){
```

```
5800            // 如果type是一个Function, 或者
    type不是function而是一个数组,那就要进行一些参数的" 矫正 "工作
5801            if ( jQuery.isFunction(type) || ( type && type.constructor ==
     Array )) {
5802                fn = type;// 让第二个参数fn依然是function,
    这样下面的程序逻辑就不用修改
5803                type = "fx";// 动画的类型为fx.
5804            }
5805
5806            // 如果没有传入type, 或者传入了type( 它是一个string
    ),但没有第二个参数,
    那么说明queue函数的调用者想获得匹配元素集合中首元素的动画
5807            // 队列. OK, 返回给他/她.
5808            if ( !type || (typeof type == "string" && !fn) )
5809                // 获得首个匹配元素的动画队列. this
    是一个jQuery对象,它含有好多的匹配元素(
    这些匹配元素都是通过selector匹配而来 ).那么这个queue函数返回的
5810                // 就是第一个匹配元素this[ 0 ]的动画队列.
5811                return queue( this[0], type );
5812
5813            // 函数如果能运行到这里,说明要设置动画队列的值. 注意 "
    return this.each(...) "的" this " 是一个 jQuery的引用,而下面的" this
    " 则是一个具体的
5814            // 匹配元素的引用.
5815            return this.each(function(){
5816                // 如果传了一个数组进来( fn是一个数组
    ),意思是要用这个数组代替元素原来的那个,成为新的动画函数队列
5817                if ( fn.constructor == Array )
5818                    queue(this, type, fn);//设置新队列
5819
5820                //
    如果仅仅是传进一个函数的引用,就把这个引用追加到动画函数队列里面
5821                else {
5822                    queue(this, type).push( fn );
5823
5824                    // 如果元素原来并没有动画,现在有了( length == 1 ),
    那马上运行动画
5825                    if ( queue(this, type).length == 1 )
5826                        fn.call(this);
5827                }
5828            });
5829        },
5830
5831        /**
5832         *
5833         * @param {Object} clearQueue
5834         * @param {Object} gotoEnd
5835         */
5836        stop: function(clearQueue, gotoEnd){
5837            var timers = jQuery.timers;
5838
5839            // 清空jQeury对象内每一个元素上的动画函数队列.
    注意,每一个jQuery对象包含若干的元素,而每一元素都含有一个动画函数队列
5840            // 还要值得注意的是,纵使你删除了元素上的动画队列,
    但是说不定队列上的某些动画函数已经进入的执行队列( jQuery.timers
    ),正在等待执行.
5841            if (clearQueue)
```

```
5842                this.queue([]);
    //把匹配元素集合中的每一个元素的动画队列置空
5843
5844            this.each(function(){
5845                // go in reverse order so anything added to the queue
    during the loop is ignored
5846                // 倒着来访问数组,从队列尾部开始删除属于this的动画函数,
    以防止后来加入队列的this的动画函数被执行-------> 这是一个好主意.
5847                for ( var i = timers.length - 1; i >= 0; i-- )
5848                    if ( timers[i].elem == this ) {
    //在执行队列中发现有属于当前元素的动画, 删除这个动画.
5849                        if (gotoEnd)
5850                            // force the next step to be the last 翻译:
    让下一步成为最后步...
5851                            //
    马上执行队列中属于this的最后一个动画函数(因为我们是从队列的尾部开始删
    除this的动画函数的), 动画马上到停止在
5852                            // 执行队列的最后一个属于this的动画函数上.
5853                            //
    而传入true给动画函数,这个true最终会交到step()手上,做为它的输入参数
5854                            // step 接收到true,
    马上将属性设置成为末尾状态.
    动画函数最终将被移出运行队列:timers.splice(i, 1);
5855                            // 请详细参考jQuery.fx.step的中文注释.
5856                            timers[i](true);
5857                        timers.splice(i, 1);
5858                    }
5859            });
5860
5861            // start the next in the queue if the last step wasn't forced
5862            if (!gotoEnd)//如果没有强制要求停止,
    那就让匹配元素的动画队列出队(数据结构术语,即删除队列头部的那个元素),
    这样动画就能继续播放.
5863                this.dequeue();
5864
5865            return this;
5866        }
5867
5868    });
5869
5870
5871    //------------------------------------------------
    定义两个函数让jQuery对象来管理自己的 动画队列
    ----------------------------
5872     /**
5873      *
    jQuery内部使用的queue函数,对外公开的queue函数实际上是使用了这个函数来
    完成任务的.
5874      * 它的作用是取得/设置储存在元素上的动画函数队列, 并将该队列返回.
5875      *
5876      * @param {HTMLElement} elem -
    传入这个参数是想elem这个元素上的queue列表.
5877      * @param {string} type - 动画函数队列的类型,如" fx ".
5878      * @param {Array} array - 动画函数队列.
    它是一个数组,如果元素本身并没有动画函数队列的时候,这个数组就会被设置成
    为元素的动画函数队列
5879      */
5880    var queue = function( elem, type, array ) {
```

```
5881        if ( elem ){
5882
5883            type = type || "fx";
5884
5885            var q = jQuery.data( elem, type + "queue" );
    //获取jQuery对象的fx动画队列
5886
5887            if ( !q || array )// 如果没有动画队列,
    或者传入了第三个参数array, 那就用array代替原来的动画队列.
5888                q = jQuery.data( elem, type + "queue", jQuery.makeArray(
    array) );
5889
5890        }
5891        return q;//返回elem的动画队列.
5892    };
5893
5894    //---------------------------------------------------
    为jQuery对象添加dequeue功能-------------------------------------
5895    /**
5896     * 从动画函数队列中删除第一个动画效果函数.
5897     * @param {string} type - 动画函数类型
5898     */
5899    jQuery.fn.dequeue = function(type){
5900        // 默认删除 fx 类的动画队列
5901        type = type || "fx";
5902
5903        return this.each(function(){
5904            var q = queue(this, type);//获取元素的动画函数队列
5905
5906            // 移除动画函数队列中第一个元素
5907            q.shift();
5908
5909            // 如果动画函数队列还有函数在里边,
    就以this作为该队列第一个元素( 它是一个函数 )的上下文来执行这个函数
5910            // 通俗点说就在shift后把队列里第一个元素q[ 0
    ]放到this里面执行.实际的效果就是删除第一个效果函数之后,继续执行下面的
    效果函数,不要让它停下来
5911            if ( q.length )
5912                q[0].call( this );
5913        });
5914    };
5915    //---------------------------------------------------------------
    ---------------------------------------------------------------
5916
5917
5918
5919
5920
5921
5922    //
    ---------------------------------------------------------------
     让jQuery对象 具有动画的能力 ---------------------------------
5923    jQuery.extend({
5924
5925        speed: function(speed, easing, fn) {
5926            var opt = speed && speed.constructor == Object ? speed : {
5927                complete: fn || !fn && easing ||jQuery.isFunction( speed
    ) && speed,
```

```
5928            duration: speed,
5929            easing: fn && easing || easing && easing.constructor !=
    Function && easing
5930
5931            /* 对上面的代码进行一点说明:
5932             * opt有三个属性:
5933             * complete : 动画完成时所调用的函数
5934             * duration : 动画持续的时长
5935             * easing : 动画效果
5936             */
5937
5938        };
5939
5940        //
    动画的持续事件.如果传入的持续时间参数duration是一个Number,
    那么就用这个数字来设置动画的时长.
5941        //
    如果传入的duration不是Number(这个时候就是string),那就到fx的动画时间类
    型(jQuery.fx.speeds)中查找到底这种duration的值
5942        // 对应的是一个什么数字.
    如果有就用上这个数字(比如说"slow"对应的数字就是600),
    没有就用默认的jQuery.fx.def(400)来代替.
5943        opt.duration = (opt.duration && opt.duration.constructor ==
    Number ?
5944            opt.duration :
5945            jQuery.fx.speeds[opt.duration]) || jQuery.fx.speeds.def;
5946
5947        // Queueing
5948        opt.old = opt.complete;//保存opt.complete到opt.old中
5949        opt.complete = function(){// 重新定义opt.complete,
    从新定义的函数相对于原来那个函数的一个"外壳"或"包裹". 可以看到重新定义
5950                                  //
    的函数内部还是调用了原来的函数(opt.old.call(this)),只不过在调用老的函
    数之前检查一下是否需要
5951                                  // dequeue.
    这种通过"包裹"或者"外壳"来扩展原函数的功能的方法在jQuery中十分常见.
5952
5953            if ( opt.queue !== false )
    //在调用old的函数之前先看看是否要dequeue.
5954                jQuery(this).dequeue();
5955            if ( jQuery.isFunction( opt.old ) )
5956                opt.old.call( this );
5957        };
5958
5959        return opt;
5960    },
5961
5962
5963    /*
5964     *
    按照一定的方程产生下一个数字.其实是想模拟特定的函数增长规律,使动画呈现
    一定的效果( 线性增长[linear]或者余弦摆动[swing] )
5965     * p - 变量每次输入进来的新值
5966     * n - 目前还没有用
5967     * firstNum - 常量
5968     * diff - 系数
5969     */
5970    easing: {
```

```
5971            linear: function( p, n, firstNum, diff ) {
5972                return firstNum + diff * p;
5973            },
5974            swing: function( p, n, firstNum, diff ) {
5975                return ((-Math.cos(p*Math.PI)/2) + 0.5) * diff + firstNum;
5976            }
5977        },
5978
5979        timers: [],
5980        timerId: null,
5981
5982        /*
5983         * 这个就是fx的构造函数了
5984         *
5985         */
5986        /**
5987         * 这个是fx的构造函数. 每一步的动画其实都是一个fx的对象.
5988         * @param {HTMLElement} elem  HTML元素的引用
5989         * @param {Object} options - 动画的参数
5990         * @param {Object} prop -
5991         */
5992        fx: function( elem, options, prop ){
5993            this.options = options;
5994            this.elem = elem;//注意这里的elem是一个HTML元素的引用
5995            this.prop = prop;
5996
5997            if ( !options.orig )
5998                options.orig = {};
5999        }
6000
6001    });
6002
6003
6004
6005
6006
6007    //---------------------------------------------------- fx动画模块
        ------------------------------------------------------------
6008    // 摘抄自网上的说明: jQuery FX, jQuery
        UI后的第二个子库, 强调动画效果而非UI的外观模块, 包括对象的消失、出现;
        颜色、大小、位置变换。而使用时是
6009    // 扩展原jQuery的API, 依旧那么华丽的简单。
6010    //------------------------------------------------------------------
        ------------------------------------------------------
6011    /**
6012     * jQuery为每个元素的每个要进行的动画的属性都新建一个jQuery.fx对象.
6013     */
6014    jQuery.fx.prototype = {
6015
6016        // Simple function for setting a style value
6017        /*
6018         * 更新fx的对象状态
6019         */
6020        update: function(){
6021            //哪里冒出来的options?
        答:options在fx的构造函数中被定义看上面的代码
6022            if ( this.options.step )
6023                this.options.step.call( this.elem, this.now, this );
```

```
6024
6025                                                    //_default:
    function(fx){
6026                                                    //
    fx.elem.style[ fx.prop ] = fx.now + fx.unit;
6027                                                    //}
6028            (jQuery.fx.step[this.prop] || jQuery.fx.step._default)( this
    );// this指的是一个jQuery.fx对象. update函数实际上是
6029
     // 调用了step中的函数来完成update的功能的. step负责
6030
     // 将计算出来的下一个属性值更新到HTML元素上. 可以说,
6031
     // 举个例子: 设this.prop == "opacity", 这样update
6032
     // 函数在这里就调用了jQuery.fx.step.opacity(this).
6033
     // opacity函数就会将当前计算到的opacity值更新到fx对象上
6034


6035
6036            // Set display property to block for height/width animations
6037            // 如果要进行 height 或者 width
    的动画,那么要将它的display样式设置成为 block.
6038            if ( this.prop == "height" || this.prop == "width" )
6039                this.elem.style.display = "block";
6040        },
6041
6042        // Get the current size
6043        /**
6044         *
6045         * @param {boolean} force - 获取元素当前属性时, 是要内联的样式,
    还是要最终计算样式(最终叠加到元素上,元素所呈现的样式).
6046         *                          可用看到, 这个属性值,
    最终是传给jQuery.css函数的. 可以参考jQuery.css函数的中文注释.
6047         */
6048        cur: function(force){
6049            // 如果this.prop是一个HTML属性, 将这个属性的值返回
6050            if ( this.elem[this.prop] != null && this.elem.style[this.
    prop] == null )
6051                return this.elem[ this.prop ];
6052
6053            // 更多的时候, this.prop是一个style属性.
    使用jQuery.css函数来获取元素的css值.
6054            var r = parseFloat(jQuery.css(this.elem, this.prop, force));
6055
6056            // 如果r有值, 并且r的值在一个可以接受的范围(>-10000),
    那就直接返回这个值; 若否, 那就调用更底层的curCSS函数来获取
6057            // 所需元素的层叠样式值. css函数内部调用了curCSS,
    基本上所有的style属性都能从css函数手上传到curCSS函数中处理, 但
6058            // 有一点例外,
    那就是当this.prop的值为"width"或"height"的时候,
    css函数自己处理过后就将结果返回, 而并没有经过curCSS
6059            // 的处理. 下面的这行代码是处于对css函数处理结果的"不放心",
    当css处理结果不符合要求的时候, 调用更底层的curCSS 来尝试处理.
6060            // 如果还不行, 那没有办法了,
    只能返回0.
6061            return r && r > -10000 ? r : parseFloat(jQuery.curCSS(this.
```

```
elem, this.prop)) || 0;
6062        },
6063
6064        // Start an animation from one number to another
6065        /**
6066         * 将数组elems内的每一个元素使用callback进行处理,
    将处理过后的元素放到一个新的数组当中, 最后返回这个数组.
6067         *
6068         * 注意: 这个函数在功能上与jQuery.map重复, 因此在jQuery
    1.3.2版中已将其删除.
6069         *
6070         * @param {Array} elems - 需要处理的元素组成的数组
6071         * @param {Function} callback - 处理函数
6072         */
6073        map: function( elems, callback ) {
6074            var ret = [];
6075
6076            // Go through the array, translating each of the items to
    their
6077            // new value (or values).
6078            for ( var i = 0, length = elems.length; i < length; i++ ) {
6079                var value = callback( elems[ i ], i );
6080
6081                // value不是 null, 说明处理成功, 把它加入新的数组里面去
6082                if ( value != null )
6083                    ret[ ret.length ] = value;
6084            }
6085
6086            //
    返回一个新的数组, 这个数组中的每个元素都是由原来数组中的元素经callback
    处理后得来
6087            return ret.concat.apply( [], ret );
6088        },
6089
6090        //jQuery 对象的animate 函数就是调用这个方法来完成最后的动画的
6091        /**
6092         * 执行一个属性值从from到to的动画.
6093         * 它是animate的底层实现.
6094         *
6095         * @param {Object} from - 属性开始值
6096         * @param {Object} to - 属性中止值
6097         * @param {Object} unit - 属性值的单位,如 " px "
6098         */
6099        custom: function(from, to, unit){
6100            this.startTime = now();
6101            this.start = from;
6102            this.end = to;
6103            this.unit = unit || this.unit || "px";
6104            this.now = this.start;
6105            this.pos = this.state = 0;
6106
6107            //更新属性设置,应用上面的设置
6108            this.update();
6109
6110            var self = this;//this指的是一个jQuery.fx动画对象.
6111
6112            //定义一个内部函数t, 这个t函数将会在执行队列里头排队等待执行.
6113            function t(gotoEnd){
```

```
6114
        //custom函数是使用setInterval不断地运行来达到动画的效果
6115                return self.step(gotoEnd);
6116            }
6117
6118            t.elem = this.elem;//注意,this.elem是一个jQuery对象
6119
6120            // 把动画函数放进动画执行队列里面了
6121            jQuery.timers.push(t);
6122
6123            // 如果整个jQuery还没有建立起计时器, 那就新建一个,
        并且一个就够了.
6124            if ( jQuery.timerId == null ) {
6125                //获取interval的引用,
        在队列中没有了要执行的函数的时候,好销毁它.
6126
        //另外使用setInterval来不断运行动画队列里面的动画函数,一直到队列里面没
        有了函数才停止
6127                jQuery.timerId = setInterval(function(){
6128                    var timers = jQuery.timers;
6129
6130                    //
        遍历动画运行队列里面的每一个动画函数,遍历的时候做以下动作:
6131                    // (1) 执行动画
6132                    // (2) 将不再需要循环执行的动画函数清出运行队列
6133                    // 那,什么是" 不再需要循环执行的动画函数 " 呢? 答:
        动画函数返回false, 以下" if(!timer[i]())timers.aplice(i--,1) "
6134                    // 就不会执行,动画函数执行完之后不会被清理出运行队列
6135                    // 那么在下一个interval里面,这个动画函数又被执行
6136                    for ( var i = 0; i < timers.length; i++ )
6137                        if ( !timers[i]() )//运行动画函数
6138                            timers.splice(i--, 1);//将动画函数清出运行队列
6139
6140
        //如果动画运行队里面没有要运行的动画函数了,清理interval
6141                    if ( !timers.length ) {
6142                        clearInterval( jQuery.timerId );
6143                        jQuery.timerId = null;
6144                    }
6145                }, 13);
6146            }
6147        },
6148
6149        // Simple 'show' function
6150        /**
6151         * 简易的元素显示
6152         */
6153        show: function(){
6154            // Remember where we started, so that we can go back to it
        later
6155            // 翻译: 记下我们是从哪个属性值开始的,
        这样我们待会需要的时候就可以重设回这个值.
6156            this.options.orig[this.prop] = jQuery.attr( this.elem.style,
        this.prop );
6157            this.options.show = true;
6158
6159            // Begin the animation 翻译: 开始动画
6160            this.custom(0, this.cur());
```

```
6161
6162            // Make sure that we start at a small width/height to avoid any
6163            // flash of content
6164            // 翻译: 确保我们从一个很小的width/height值开始动画,这样可以避免内容的闪烁.
6165            if ( this.prop == "width" || this.prop == "height" )
6166                this.elem.style[this.prop] = "1px";
6167
6168            // Start by showing the element
6169            jQuery(this.elem).show();// 调用jQuery对象的show方法.
6170        },
6171
6172        // Simple 'hide' function
6173        /**
6174         * 隐藏的动画
6175         */
6176        hide: function(){
6177            // Remember where we started, so that we can go back to it later
6178            // 翻译: 记下我们是从哪个属性值开始的,这样我们待会需要的时候就可以重设回这个值.
6179            this.options.orig[this.prop] = jQuery.attr( this.elem.style, this.prop );
6180            this.options.hide = true;
6181
6182            // Begin the animation 翻译: 开始动画
6183            this.custom(this.cur(), 0);
6184        },
6185
6186        // Each step of an animation
6187        /**
6188         * 一个动画的一步(也即每一个帧)都由这个函数来执行.
6189         *
6190         * @param {Object} gotoEnd
6191         */
6192        step: function(gotoEnd){
6193            var t = now();
6194
6195
6196            // 如果动画过期了或者强制要求动画停止(gogoEnd == true)
6197            if ( gotoEnd || t > this.options.duration + this.startTime ) {
6198                //让现在的状态马上变成末状态, 动画已经过期
6199                this.now = this.end;//this.end == custom( form, to, unit )中的 to
6200
6201                this.pos = this.state = 1;//在custom函数里面,他们都是0
6202                this.update();//更新状态
6203
6204                //this.options.curAninm内装的是用户要设置的属性的集合
6205                //把这个属性设置为true, 表示动画已经完成
6206                this.options.curAnim[ this.prop ] = true;
6207
6208                var done = true;
6209                for ( var i in this.options.curAnim )
6210                    if ( this.options.curAnim[i] !== true )
//如果有一个属性的动画没有完成,都不算是全部完成,done = false;
6211                        done = false;
```

```
6212
6213
//能运行到这里,表示curAnim[]数组里面全是true,那意味着用户设置的所有属性的动画都已经完成
6214            if ( done ) {
6215                if ( this.options.display != null ) {
6216                    // Reset the overflow
6217                    this.elem.style.overflow = this.options.overflow;
6218
6219                    // Reset the display
6220                    this.elem.style.display = this.options.display;
6221                    if ( jQuery.css(this.elem, "display") == "none" )
6222                        this.elem.style.display = "block";
6223                }
6224
6225                // Hide the element if the "hide" operation was done
6226                // 如果这个时候hide了( 用户调用了fx对象的hide操作),那就hide 了它
6227                if ( this.options.hide )
6228                    this.elem.style.display = "none";
6229
6230                // Reset the properties, if the item has been hidden or shown
6231                if ( this.options.hide || this.options.show )
6232                    for ( var p in this.options.curAnim )
6233                        jQuery.attr(this.elem.style, p, this.options.orig[p]);// reset操作, options里面的都是初始值
6234            }
6235
6236            if ( done )
6237                // Execute the complete function
6238                // this指向的是一个jQuery.fx对象,而this.elem则是一个HTML元素.
6239                this.options.complete.call( this.elem );
//动画函数完成, 调用需要在这个时候执行的那个callback.
6240
6241            //
注意step函数运行的时机,它在custom函数中被包裹在临时函数t内,然后被加入到jQuery.timers里面而得到运行的
6242            //
返回false,那么这个step就会在运行结束之后被清理出运行队列,在timers的下一个interval中,它( 这个step )将不会再被运行
6243            return false;
6244        }
6245
6246
6247        //动画没有过期或者没有被强制停止
6248        else {
6249            var n = t - this.startTime;//算算动画开始多久了
6250            this.state = n / this.options.duration;
//算算完成了几分之几啊
6251
6252            // Perform the easing function, defaults to swing
    //参数n,this.options.duration传给easing
6253            // 执行动画扰动函数,在默认情况
```

```
            //[...]没有效果的,因为函数体并没有使用到这些
6254
                                                                    //变量
6255            this.pos = jQuery.easing[this.options.easing || (jQuery.
            easing.swing ? "swing" : "linear")](this.state, n, 0, 1, this.options
            .duration);
6256
6257            //
            计算下一个step所使用属性值.留意一下this.pos这个值,就是这个值使得元素属
            性的变化呈现线性或者余弦摆动的特性.
6258            this.now = this.start + ((this.end - this.start) * this.
            pos);
6259
6260            // Perform the next step of the animation
6261            //
            在上面处理完下一step所有使用的属性之后,当然就是更新了.这样才有动画效果
6262            this.update();
6263        }
6264
6265        // 返回true,说明这个step完成,但是并不退出动画函数队列.
6266        //
            注意step函数运行的时机,它是在custom函数中被包裹在临时函数函数t内,然后
            被加入到jQuery.timers里面而得到运行的.
6267        //
            返回true,那么这个step就会在运行结束继续留在动画函数队列里头,等待下一个
            interval再次被执行.
6268        return true;
6269    }
6270
6271};
6272
6273
6274
6275//-----------------------设置 fx 的动画参数常量speeds 和
            定义4个基本的动画"步进(下一帧)"函数-----------------------------------
            ----------
6276Query.extend( jQuery.fx, {
6277    /**
6278     * 默认的动画速度类型
6279     * 他们的单位是毫秒
6280     */
6281    speeds:{
6282        slow: 600,
6283        fast: 200,
6284        // Default speed
6285        def: 400
6286    },
6287    /**
6288     * 设置动画"步进"方法.
6289     *
            设置元素(fx.elem)位置动画、透明度动画、或者其他属性动画的"下一帧"索要
            显示的值.
6290     * 可以看到step支部是是一个命名空间, 其内的"scrollLeft",
            "scrollTop"等才是真正设置"下一帧"所需属性值的"步进"函数.
6291     * 如果你还是不明白我在讲什么, 没有关系, 代码并不复杂:
6292     * @param {jQuery.fx} fx - 当前的动画对象.
            jQuery为每一个需要进行动画的属性都创建一个动画对象.
6293     */
```

```
6294        step: {
6295            /**
6296             * 把元素定位到当前动画所需要的位置(水平方向)
6297             * @param {jQuery.fx} fx - 当前的动画对象.
    jQuery为每一个需要进行动画的属性都创建一个动画对象.
6298             */
6299            scrollLeft: function(fx){
6300                fx.elem.scrollLeft = fx.now;
6301            },
6302
6303            /**
6304             * 把元素定位到当前动画所需要的位置(垂直方向)
6305             * @param {jQuery.fx} fx - 当前的动画对象.
    jQuery为每一个需要进行动画的属性都创建一个动画对象.
6306             */
6307            scrollTop: function(fx){
6308                fx.elem.scrollTop = fx.now;
6309            },
6310
6311            /**
6312             * 把元素的opacity属性的值设置到当前动画所需要的值
6313             * @param {jQuery.fx} fx - 当前的动画对象.
    jQuery为每一个需要进行动画的属性都创建一个动画对象.
6314             */
6315            opacity: function(fx){
6316                jQuery.attr(fx.elem.style, "opacity", fx.now);
6317            },
6318
6319            /**
6320             *
    把元素的属性(这个属性由fx.prop指定)值设置到当前动画所需的值.
6321             * @param {jQuery.fx} fx - 当前的动画对象.
    jQuery为每一个需要进行动画的属性都创建一个动画对象.
6322             */
6323            _default: function(fx){
6324                fx.elem.style[ fx.prop ] = fx.now + fx.unit;
6325            }
6326        }
6327    });
6328
6329
6330    // The Offset Method
6331    // Originally By Brandon Aaron, part of the Dimension Plugin
6332    // http://jquery.com/plugins/project/dimensions
6333    /**
6334     * 获取匹配元素集合中首元素的offset.
6335     * 所谓offset是元素在文档中的坐标.
6336     */
6337    jQuery.fn.offset = function() {
6338                                            //this[0]表示jQuery
    选择器匹配的所有元素中的第一个元素
6339
    //可见这个函数只是对jQuery选择器选中的第一元素器作用
6340        var left = 0, top = 0, elem = this[0], results;
6341
6342        // 如果这个元素存在, 在jQuery.browser 命名空间下做些事情
6343        // with相当于using namespace, 是在某个命名空间下,
    这样就省得要在调用函数时写一大串的名字空间前缀.
```

```
6344        if ( elem ) with ( jQuery.browser ) {
6345            var parent       = elem.parentNode,
6346                offsetChild  = elem,
6347                offsetParent = elem.offsetParent,
6348                doc          = elem.ownerDocument,//
    取得某个节点的根元素( document对象 )
6349                safari2      = safari && parseInt(version) < 522 && !
    /adobeair/i.test(userAgent),
6350                css          = jQuery.curCSS,
6351                fixed        = css(elem, "position") == "fixed"; //
    元素是否是fixed定位.
6352
6353            // Use getBoundingClientRect if available
6354            // getBoundingClientRect 是IE 的方法
6355            if ( elem.getBoundingClientRect ) {
6356                var box = elem.getBoundingClientRect();//
    获得与元素绑定的客户区矩形,
    通过这个矩形就能获取元素相对与document的坐标.
6357
6358                // Add the document scroll offsets
6359                // 可能document也发生了偏移,
    因此把document的offset也算上, 万无一失.
6360                // 这里的add函数是一个内部定义的函数,
    它的功能主要将它的两个参数分别累加到left和top上.
6361                add(box.left + Math.max(doc.documentElement.scrollLeft,
    doc.body.scrollLeft),
6362                    box.top  + Math.max(doc.documentElement.scrollTop,
    doc.body.scrollTop));
6363
6364                // IE adds the HTML element's border, by default it is
    medium which is 2px
6365                // IE 6 and 7 quirks mode the border width is
    overwritable by the following css html { border: 0; }
6366                // IE 7 standards mode, the border is always 2px
6367                // This border/offset is typically represented by the
    clientLeft and clientTop properties
6368                // However, in IE6 and 7 quirks mode the clientLeft and
    clientTop properties are not updated when overwriting it via CSS
6369                // Therefore this method will be off by 2px in IE while
    in quirksmode
6370                /* 翻译: IE会加上HTML元素的边框,
    在默认情况之下为medium也就是2px
6371                 * IE 6 和 7 的怪癖模式中这个HTML元素上的border
    width可以用css规则"html{ border:0}"来重写
6372                 * 在IE7的标准模式中, 这个边框宽度总是2px
6373                 * 这个border/offset
    在一般情况下还可由clientLeft和clientTop属性表示(因为这个border导致了HT
    ML元素的偏移—译者注)
6374                 * 然而,
    在IE6和7的怪癖模式中修改clientLeft和clientTop属性的值并不能修改这个bor
    der
6375                 *
    因此那这种方法(下面这行代码所用的方法—译者注)将会在IE的怪癖模式中失效
    .
6376                 */
6377                add( -doc.documentElement.clientLeft, -doc.
    documentElement.clientTop );//剪掉IE给HTML所添加的border
6378
```

```
6379            // Otherwise loop through the offsetParents and parentNodes
6380            }
6381
6382            // 不是IE 浏览器的话...那就是其他符合w3c标准的浏览器
6383            else {
6384
6385                // Initial element offsets 翻译:初始化元素的offsets
6386                /* offsetLeft和offsetTop是元素相对于最近定位祖先的偏移量.
6387                 *
        最近定位祖先就是元素的祖先中最近的,并设置了postion为fixed/absolute/rel
        ative的元素.
6388                 * 如果没有这样的元素, "最近定位祖先"就是document.
6389                 */
6390                add( elem.offsetLeft, elem.offsetTop );
6391
6392                // Get parent offsets
6393                // 获取parent的offset, 防止元素的parent也发生了偏移.
6394                while ( offsetParent ) {
6395                    // Add offsetParent offsets
        翻译:把offsetParent的offset也加进来.
6396                    add( offsetParent.offsetLeft, offsetParent.offsetTop
        );
6397
6398                    // Mozilla and Safari > 2 does not include the
        border on offset parents
6399                    // However Mozilla adds the border for table or
        table cells
6400                    /* 翻译: Mozilla和Safari
        2以上的浏览器在计算offsetParent的offsets的时候并没有把border也算进去,
6401                     * 不过 Mozilla会给table和table
        单元格(即td,th)算上border的宽度.
6402                     */
6403                    if ( mozilla && !/^t(able|d|h)$/i.test(offsetParent.
        tagName) || safari && !safari2 )
6404                        border( offsetParent );//加上offsetParent 的border
6405
6406                    // Add the document scroll offsets if position is
        fixed on any offsetParent
6407                    // 如果元素的offsetParent是position:fixed的,
        那么加上视窗的滚动量.
6408                    if ( !fixed && css(offsetParent, "position") ==
        "fixed")
6409                        fixed = true;
6410
6411
6412                    /*
6413                     *
        下面的两行代码设置offetChild/offsetParent为新的值,
        这样在新一轮的循环中我们才能通过他们计算出正确的left和top的值.
6414                     */
6415
6416
6417                    // Set offsetChild to previous offsetParent unless
        it is the body element
6418                    // 翻译: 设置offsetChild为当前offsetParent,
        除非它(即当前的offestChild)就是body.
6419                    offsetChild  = /^body$/i.test(offsetParent.tagName) ?
         offsetChild : offsetParent;
```

```
6420
6421
6422                    // Get next offsetParent
6423                    // 继续往上层看, 如果还有offetParent
    就继续加他们的offsetLeft和offsetTop和boerderWidth加进来
6424                    offsetParent = offsetParent.offsetParent;
6425                }
6426
6427
6428                // Get parent scroll offsets
6429                // 如果浏览器窗口并没有产生滚动的时候,
    前面的代码已经能完成任务, 但是如果窗口发生了滚动,
    单单是前面的代码就不够用了.
6430                //
    所以在获取完parent的offset之后,接下来就要获取parent的scroll了.
6431                //
6432                // scroll是在父元素出现滚动条的情况之下才会发挥作用
6433
6434                // parent存在并且这个parent
    不是body或者html,那么我们就一直向上遍历元素的祖先,并且在这个过程当中如
    果发现某个祖先出现了滚动条
6435                // 就减去这个条所产生的滚动量.
6436                while ( parent && parent.tagName && !/^body|html$/i.test(
    parent.tagName) ) {
6437                    /* Remove parent scroll UNLESS that parent is inline
    or a table to work around Opera inline/table scrollLeft/Top bug
6438                     * COMP:删除 parent 的 scroll 除非parent
    是inline元素或者table. 这样做的目的是为了处理 Opera
    行内元素和table的scrollLeft/Top
6439                     * 的bug.
6440                     *
6441                     * 在两种情况之下会出现滚动条:
6442                     * (1) 窗口不足以显示整个document
6443                     * (2) 元素设置了overflow:auto或者overflow:scroll
6444                     *
6445                     * 因此,
    这样遍历一遍所有的parentNode并减去parent出现的scroll是必要的.不然的话,
    我们直接减去document的scroll就可以了.
6446                     */
6447                    if ( !/^inline|table.*$/i.test(css(parent, "display"
    )) )
6448                        // Subtract parent scroll offsets
6449                        add( -parent.scrollLeft, -parent.scrollTop );
6450
6451                    // Mozilla does not add the border for a parent that
    has overflow != visible
6452                    // COMP:翻译:
    Mozilla在计算设置了overflow!=visible的parent的offset的时候,
    并没有把border也算进去.
6453                    /* 解释一下:
6454                     * 使用scrollLeft/Top来获取一个元素的滚动量的时候,
    这个offset是相对border的外边缘计算的. 那么当元素具有了非0边框的时候,
6455                     *
    offsetLeft/Top就应该包含边框的宽度.但是Mozilla在计算overflow!=visible
    的元素的scrollLeft/Top的时候,并没有把这个
6456                     *
    border算进去,即相当于相对于边框的内边缘计算,而忽略了border的宽度,
    于是jQuery就使用以下的代码把这种情况下缺失的border
```

```
6457                 * 算进去.
6458                 */
6459                if ( mozilla && css(parent, "overflow") != "visible" )
6460                    border( parent );
6461
6462                // Get next parent
6463                parent = parent.parentNode;//获取下一个parent.
6464            }
6465
6466            // Safari <= 2 doubles body offsets with a fixed
    position element/offsetParent or absolutely positioned offsetChild
6467            // Mozilla doubles body offsets with a non-absolutely
    positioned offsetChild
6468            // COMP:safari 2 以下的浏览器的一个bug:
6469            // 如果所求元素的position=="absolute"或者为"fixed",
    那么safari会把body的offsets(Left/Top)算多一倍.(要减掉这一倍)
6470            // 另外Mozilla(Firefox)也有一个类似的bug:
6471            // 当所求元素的position != "absolute" 时, body
    的offsets也会被算多一倍.(要减掉这一倍)
6472            if ( (safari2 && (fixed || css(offsetChild, "position")
    == "absolute")) ||
6473                (mozilla && css(offsetChild, "position") !=
    "absolute") )
6474                    add( -doc.body.offsetLeft, -doc.body.offsetTop );
6475
6476            // Add the document scroll offsets if position is fixed
6477            //
    翻译:当元素是postion:fixed的时候,就把document的scroll也添加上去
6478            if ( fixed )      //doc = ownerDocument
6479                add(Math.max(doc.documentElement.scrollLeft, doc.body
    .scrollLeft),
6480                    Math.max(doc.documentElement.scrollTop,  doc.body
    .scrollTop));
6481        }
6482
6483        // Return an object with top and left properties
6484        // 用一个对象将offset计算结果保存,待会返回.
6485        results = { top: top, left: left };
6486    }
6487    /**
6488     *
    获取元素的borderLeftWidth和borderTopWidth并把他们分别叠加到left和top中
    去.
6489     * @param {HTMLElement} elem - HTMLElement
6490     */
6491    function border(elem) {
6492        add( jQuery.curCSS(elem, "borderLeftWidth", true), jQuery.
    curCSS(elem, "borderTopWidth", true) );
6493    }
6494
6495    /**
6496     * 将l和t分别加入到left和top中
6497     * @param {Number} l
6498     * @param {Number} t
6499     */
6500    function add(l, t) {
6501                                //10表示十进制
6502        left += parseInt(l, 10) || 0;
```

```
6503            top += parseInt(t, 10) || 0;
6504        }
6505
6506        return results;
6507
6508    };//jQuery.fn.offset函数.
6509
6510
6511
6512    jQuery.fn.extend({
6513        /**
6514         *
        计算元素相对于自己的offsetParent的相对偏移.函数返回一个对象,
        对象内有left, top两个属性分别存储着元素相对于自己的offsetParent的偏移.
6515         *
6516         *  请注意,
        position函数使用了子元素的offset与父元素的offset作差的方式来获得一个元
        素相对于它的父亲元素的偏移.
6517         *  在jQuery.fn.offset函数中我们可以看到,
        直接使用元素offsetLeft/Top来获取元素相对于最近定位祖先的偏移会存在风险
        .因为元素可能会处在一个
6518         *
        产生了滚动条的容器当中.因此我们在这里统一使用jQuery.fn.offset方法来获
        取容器元素和子元素的offset,然后两个offset作差, 这样获得的offset
6519         *  就是比较准确的偏移.
6520         *
6521         *  同时也注意,
        自己的offsetParent不一定是在Dom中将自己包含的容器元素.
        offsetParent应该是最近的已定位(设置了postion:fixed/absolute/relative)
6522         *  祖先.
6523         */
6524        position: function() {
6525            var left = 0, top = 0, results;
6526
6527            // 注意哦,this指向的是一个jQuery对象.
        在这里可以看到jQuery.fn.position函数只获取匹配元素集合中首元素的positi
        on
6528            if ( this[0] ) {
6529                // Get *real* offsetParent
6530                // 使用offsetParent()获取元素真正的offsetParent.
6531                var offsetParent = this.offsetParent(),
6532
6533                // Get correct offsets
6534                //
        offset是一个键/值对的集合:{left:someVlaue,top:someValue},jQuery对象的o
        ffset方法能够准确地,跨浏览器地获取元素的offset
6535                offset       = this.offset(),
6536
6537                //
        如果offsetParent是body或者是html就让它的偏移(parentOffset)为{top:0,lef
        t:0},因为他们都没有offsetParent了.
6538                //
        如果是一般的元素(即非(body或html))否则就使用offset()来计算咯.
6539                parentOffset = /^body|html$/i.test(offsetParent[0].
        tagName) ? { top: 0, left: 0 } : offsetParent.offset();
6540
6541                // Subtract element margins
6542                // 接下来就要减去element的margins
```

```
6543
6544            // note: when an element has margin: auto the offsetLeft
    and marginLeft
6545            // are the same in Safari causing offset.left to
    incorrectly be 0
6546            //  在 Safari中,如果元素被设置成 margin:auto,
    那么元素的offsetLeft和
    marginLeft就会变成一样,并且会导致offset.left错误地变为0.
6547            offset.top  -= num( this, 'marginTop' );
    //函数获取this[0]元素的'marginTop'的属性值的数字部分,下同.
6548            offset.left -= num( this, 'marginLeft' );
6549
6550            // Add offsetParent borders
6551            //
    子元素的定位是相对于最近定位祖先border的内边缘来计算的,
    而parentOffset所获得的最近定位祖先的偏移并没有包括最近定位祖先的border
6552            // 的宽度.大家可以画一个图就可以明白,
    如果直接使用两个offset相减得到的结果将会多了定位祖先border的宽度.因此
    在这里将这个border的宽度
6553            // 将到parentOffset里面去,
    待会作差的时候(它是减数)就会将这个border的宽度减去.
6554            parentOffset.top  += num( offsetParent, 'borderTopWidth'
    );
6555            parentOffset.left += num( offsetParent, 'borderLeftWidth'
     );
6556
6557            // Subtract the two offsets
6558            //
    使用子元素的offset减去父亲元素的offset来获得子元素相对于父亲元素的偏移
    .
6559            results = {
6560                      //子元素相对于当前视口的偏移
6561                top:  offset.top  - parentOffset.top,
6562                                    //父元素相对于当前视口的偏移
6563
6564                // 算出来的这个就是子元素相对于父元素的偏移
6565                // 下同
6566
6567                left: offset.left - parentOffset.left
6568            };
6569        }
6570
6571        return results;//最后把结果返回.
6572    },
6573
6574    /**
6575     * 获取匹配元素集合中首元素的offsetParent,即最近定位元素
6576     */
6577    offsetParent: function() {
6578        var offsetParent = this[0].offsetParent;
6579
6580        //while循环作用是,只要元素的offsetParent不是body或者html,
    那么一直向上追溯它的最近定位祖先(即position为absolute/fixed/relative的
    祖先).
6581                                    //
    offsetParent不是body或者html             // offsetParent的position
    == 'static'
6582        while ( offsetParent && (!/^body|html$/i.test(offsetParent.
```

```
tagName) && jQuery.css(offsetParent, 'position') == 'static') )
6583                offsetParent = offsetParent.offsetParent;
6584
6585            return jQuery(offsetParent);//用jQuery对象将结果返回
6586        }
6587    });
6588
6589
6590    // Create scrollLeft and scrollTop methods
6591    /*
6592     * 下面定义两个函数scrollLeft和scrollTop:
6593     * scrollLeft计算水平方向上的滚动量/或者将页面滚动到指定的位置
6594     * scrollTop计算竖直方向上的滚动量/或者将页面滚动到指定的位置
6595     *
6596     *
    这些函数仅仅针对匹配元素集合中的首元素,并且这些首元素也是有条件的,
    即它必须是window或者是document.
6597     */
6598    jQuery.each( ['Left', 'Top'], function(i, name) {
6599        var method = 'scroll' + name;
6600
6601        jQuery.fn[ method ] = function(val) {
6602            if (!this[0]) return;
6603
6604            return val != undefined ?
6605
6606                // Set the scroll offset
6607                // 如果有给函数传入一个值val,那就scroll 到val这个位置上
6608                this.each(function() {
6609                    this == window || this == document ?
6610                        // i 表示数组['Left',
    'Top']中元素的下标.可以看到,这个下标不是0就是1
6611                        // 当 i 是 0 时,说明正在设置Left的值
6612                        // window将会scroll到这个坐标:(val,原来的Top值);
6613
6614                        // 当 i 为 1 时,说明正在设置Top的值
6615                        // window将会scroll到这个坐标:(原来的Left,val);
6616                        window.scrollTo(
6617                            !i ? val : jQuery(window).scrollLeft(),
6618                             i ? val : jQuery(window).scrollTop()
6619                        ) :
6620
    //如果正在设置的元素不是window或者document,那么仅仅是把值设置上去,不用
    做出实际的行动
6621                        this[ method ] = val;
6622                }) :
6623
6624                // Return the scroll offset
6625                //
    如果没有给函数传入val这个值,那么表明是要获得这些属性的值(val !=
    undefined返回false)
6626                this[0] == window || this[0] == document?
6627                    /* i 的作用跟上面一样, 0 就是scrollLeft; 1
    就是scrollTop
6628                     * self指向当前的window.
6629                     *
    下面三行代码首先尝试w3c标准的方法来获取窗口的滚动量,这一般是针对非IE的
    现代浏览器
```

```
6630                     *
如果不行则尝试在document.documentElement上获取,这主要针对的是IE6在解析以<DOCTYPE>开头的文档时的情况.
6631                     * 如果还是不能获取,
则使用document.body[methed]的方式来获取窗口的滚动量,
这主要是针对IE4-5以及IE6的怪癖模式.
6632                     */
6633                    self[ i ? 'pageYOffset' : 'pageXOffset' ]||
//pageX/YOffset是netscape的方法.
6634
6635                    jQuery.boxModel && document.documentElement[ method ]||//documentElement是一个快捷方式,用在IE6/7的strict模式中
6636
 // boxModel是一个boolean,表示是否在使用w3c的盒子模型
6637
6638                    document.body[ method ]:
6639
6640                    this[0][ method ];
6641        };
6642    });
6643
6644    // Create innerHeight, innerWidth, outerHeight and outerWidth methods
6645    /*
6646     * 创建innerHeight, innerWidth, outerHeight 和 outerWidth方法
6647     * inner例解:innerWidth = width(内容宽度) + paddingLeft + paddingRight
6648     * outer例解:outerWidth = innerWidth + borderLeftWidth + borderRightWidth + marginLeftWidth + marginRightWidth
6649     * (innerHeight和outerHeight的算法同上)
6650     * 下面的程序就是按照上述的方法计算的.
6651     */
6652    jQuery.each([ "Height", "Width" ], function(i, name){
6653
6654        var tl = i ? "Left"  : "Top",  // top or left
6655            br = i ? "Right" : "Bottom"; // bottom or right
6656
6657        // innerHeight and innerWidth
6658        jQuery.fn["inner" + name] = function(){
6659                    //this是一个jQuery对象
6660                    //内容的Height/Width
6661            return this[ name.toLowerCase() ]() +
6662                //paddingLeft和paddingRight(padingTop和paddingBottom)
6663                num(this, "padding" + tl) +
6664                num(this, "padding" + br);
6665        };
6666
6667        // outerHeight and outerWidth
6668        // outer就是inner再加上border和margin
6669        jQuery.fn["outer" + name] = function(margin) {
//name要么是Height,要么是Width
6670            return this["inner" + name]() +
6671                num(this, "border" + tl + "Width") +//tl为"Left" or "Top"
6672                num(this, "border" + br + "Width") +//br为"right" or "bottom"
6673                (margin ?
6674                    num(this, "margin" + tl) + num(this, "margin" + br) : 0);
6675        };
6676
```

```
6677 });})();
6678
```