Jquery 源码分析

1、概述

jQuery 是一个非常优秀的 JS 库，与 Prototype,YUI,Mootools 等众多的 Js 类库相比，它剑走偏锋，从 web 开发的实用角度出发，抛除了其它 Lib 中一些中看但不实用的东西，为开发者提供了优美短小而精悍的类库。其使用简单，文档丰富，而且性能高效，能极大地提高 web 系统的开发效率。因此可以说是 web 应用开发中最佳的 Js 辅助类库之一。大部分开发者正在抛弃 Prototype，而选择 Jquery 做为他们进行 web 开发的 JS 库。

如是开发人员仅仅只知道文档中的简单的使用方法，却不明白 Jquery 的运行原理和内部机制，在使用 jquery 时，肯定会碰到许多的问题。这些问题有一部分是 Jquery 的 Bug。大部分是自身的使用不当而造成的。而文档的简单的使用说明很难解决问题。在调试基于 jQuery 的 web 应用时，很多时候都要跟踪进入 jQuery 对象分析其运行状态以了解出错的原因。

如果对于 web 的应用的页面运行性能和效率有所要求的话，那么我们更应该去明白其运行机理和核心源码。但是 jQuery 源码不像其它的类库那样，它有点晦涩，难懂。这就是本源码分析的原因，让所有使用 jQuery 的读者，能快速上手 jQuery 的源码，并在开发中得心应用。

Jquery 的网络资源丰富，但 Baidu 了很久，很难找到那种完全深入地分析 Jquery 源码的文稿。倒是 Jquery 的开发者，John Resi 的《Pro Javascript Techniques》涉及到 Jquery 的源码的分析，但是其主指还是在于 JavaScript 的使用。那本书并不能使我们完全细致地了解 Jquery 的源码。

第一篇 Query（查询）

2、构建 jQuery 对象

在本节中，我们会就 Jquery 的运行机制和设计理念进行分析及说明。本节主要从 jQuery 的设计理念 e 及其构建的源码进行剖析。

2.1、jQuery 的设计理念

在使用 jQuery 之前，我们也许会问 jQuery 是什么？其实它的名字就很能反映其主旨的。J 是的 JS，Query 是指查询。如果把 jQuery 看作是一个查询的 JS 类库。它和 prototype，mootools 等类库一样，为 Web 的 Js 开发提供辅助功能。

那为什么要选用 jQuery 呢？在 jQuery 出现之前，Prototype，YUI 都已经是成熟的 Js 的框架，而且是各有各的特点。并且市场的书和使用文档都很详尽。为什么开发员会抛弃它们，而使用后起之秀的 jQuery，它有什么优秀的特性吸引开发人员呢？

回答这个问题，我们得明白 jQuery 的设计理念。回忆或想象一下，我们在 web 开发中是如何使用 JS？绝大多数时间都是进行如下五个方面的事情：

1、采用 getElementById 在 Dom 文档中找到 Dom 元素，然后取值或设值。

2、对元素采用 innerHTML 取其内容或设定其内容。

3、对元素进行事件的监听(如 click)。

4、通过改变元素的 CSS 样式如 height，达到视觉上的效果。

5、通过 Ajax 从服务器取值，往指定元素里添充内容。

从上面可以看，在使用 Js 开发时候就是在对 Dom 元素在进行操作。这个 Dom 元素可能是单个或是集合的形式。对元素元素操作，对于 document，window 是可以直接引用，但是对于其它的 Dom 元素，我们得从 Dom 文档树查找得到吧。这样的话就可以把 JS 的操作分析两部分任务，

一、查找 Dom 元素，

二、对 Dom 元素进行操作。

对于使用 JS 熟练的开发者而言，也许手写 document. getElementById 或 elment.getElementsByTagName 这样冗长的直接查找 Dom 元素觉得不是什么问

题，许会对 element 的 event,attribute,style 等操作也不含糊，但是对于 IE、mozilla 等几大主流的浏览器的兼容足够让每一个 JS 高手头疼。

这是使用 JS 类库的主要原因。JS 类库只要用得恰当的话，也不一定比直接采用 JavaScript 的原始函数和对象的运行效率低。但是其却能极大地提高开发的效率。

自从 Prototype 采用$符号做为 document. getElementById 的缩写，$符号似乎成了查找元素的代理符号。但是这种简单的查找并不能满足 web 应用的需要。很多时间我们需要像 CSS selector 那样查找 Dom 元素。

jQuery 从这里出发，采用$符号做为查找元素的代理。它不再是那种简单的 getElementById，而是功能强大的 CSS selector 的选择器。这也就是 query 的本意。解决了查找的元素的任务，之后就是对元素的操作。

jQuery 抛弃了 prototype 中那么对 Array,Object,Function，Event 等 JS 原生对象的扩展。把所有的心思都放在解决实际问题的 Dom 元素的操作上。它不仅简化 Dom 元素原生的冗长名字的函数名和众多难记的方法，而且在简化这些方法的同时提供了更为便捷且兼容浏览器的功能。同时那些实用的功能一个都没少，如 Ajax，Event，Fx，CSS 的操作应有仅有。

Prototype 中 Event，Ajax 等众多的对象不但让人觉得烦琐难记，而且让人感觉有点畏惧感。jQuery 在设计时就考虑到这一点。它提供了统一的入口，就是一个对象：jQuery 对象（$）。所有的操作，变化都是针对这个对象。

现在可能给 jQuery 一个明确的解释：jQuery 实质就是一个查询器。在查询器的基础还提供对查找到的元素进行操作的功能。这样说来 jQuery 就是查询和操作的统一。查询是入口，操作是结果。

jQuery 对象在代码分成两大部分，一部分是 jQuery 的静态方法，也可以称作实用方法或工具方法，通过 jQuery.xxx()的 jQuery 命名空间直接引用。第二部分是 jQuery 的实例方法，通过 jQuery(xx)或$(xx)来生成 jQuery 实例，然后通过这个实例来引用的方法。这部分的方法大多数是从采用静态方法代理来完成功能。真正的功能性的操作都在 jQuery 的静态方法中实现。

这些功能细分起来，可以分成以下几个部分：

1、Selector 查找元素。这个查找不但包含基于 CSS1~CSS3 的 CSS Selector 功能，还包含其对直接引用或间接引用 Dom 元素而扩展的一些功能。

2、Dom 元素的属性操作。Dom 元素可以看作 html 的标签，对于属性的操作就是对于标签的属性进行操作。这个属性操作包含增加，修改，删除，取值等。

3、Dom 元素的 CSS 操作。CSS 是控制页面的显示的效果。对 CSS 的操作

那就得包含高度，宽度，display 等这些常用的 CSS 的功能。

4、Ajax 的操作。Ajax 的功能就是异步从服务器取数据然后进行相关操作。

5、Event 的操作。对 Event 的兼容做了统一的处理。

6、动画(Fx)的操作。可以看作是 CSS 样式上的扩展。

2.2、jQuery 对象的构建

上一节分析了从整体上分析了 jQuery 的原理，从其原理可以看出，其统一的入口就是 jQuery 对象。那么这个对象是如何生成的呢？上一节还提到了 jQuery 的实质是 Query，那么生成 jQuery 对象就可能看作是构建并运行一个查询器。

既然是查询，肯定会有查找到的结果（Dom 元素），那么这些结果又存放在哪里呢？最好的地方当然是 jQuery 对象内面。查询的结果可能是单个元素，也可能是集合如 NodeSet。

也就是说 jQuery 对象内面应该有一个集合。且这个集合是用来存放查询到 Dom 元素。但 jQuery 对象是所有操作的统一入口，那么它的构建就不应只局限于从 Dom 文档树中查询到 Dom 元素，有可能是从别的集合中转移过来的 Dom 元素，或是 html 的片断生成的 Dom 元素。

Jquery 文档中提供了四种构建方式：jQuery(expression,[context])，jQuery(html)，jQuery(elements)，jQuery(callback)。其中 jQuery 可以用$代替。这四种方式是经常用到。其实 Jquery 的参数可以是任何的元素。也就是说任何的参数都可以构建 jquery 对象。举几个例子：

1、$($(“P”))可以看出其参数可以是 jQuery 对象或 ArrayLike 的集合。

2、$()是$(document)的简写。

3、$(3)会把 3 放到 jQuery 对象中集合中。

对于如$(3)这样的其中元素（如 ArrayLike 集合的元素）不是 Dom 元素，最好不要构建 jQuery 对象，jQuery 对象的方法设计的目的都是针对于 dom 对象的而进行的操作。如果不清楚其使用的话，很有可能会导致错误。

上面讲了这么多大道理，现在从源码的角度细细分析：

通过 jQuery(xxx)的调用实现没有生成对象，它的 this 是指向 window 对象的。那么 jQuery 的那些实例方法是怎样继承过来的呢？看一下：

```
var jQuery = window.jQuery = window.$ = function(selector, context) {
    return new jQuery.fn.init(selector, context);                    ①
};
```

这是 jquery 的总入口，jQuery 对象不是通过 new jQuery 来继承其 prototype 中的方法，而是 jQuery.fn.init 函数生成的对象。

这里我们可以看出对于 jQuery.prototype 添加一些函数集的对象的意义不大。new jQuery()还是可以的，但是生成的 jQuery 对象在 return 时会被抛弃。故不要用 new jQuery()来构建 jQuery 对象。

jQuery 对象其实就是 jQuery.fn.init 对象。那么 jQuery.fn.init.prototype 上就是挂着 jQuery 对象的操作方法。如

```
jQuery.fn.init.prototype = jQuery.fn;
```

有时间可能会担心在 589 行就实现了把 jQuery.fn 中的函数放到 jQuery.fn.init.prototype 上去，那么之后的通过 jquery.fn.extend 的方法怎么办呢？这里是对 jQuery.fn 的引用。在扩展 jQuery 的时候，只要把相关的函数 extend 到 jQuery.fn 就可以了。

现在我们看一下 jQuery.fn.init 是怎么完成工作的：

```
init : function(selector, context) {
        selector = selector || document;// 确定selector存在
        // 第一种情况 Handle $(DOMElement)单个Dom 元素，忽略上下文
        if (selector.nodeType) {                                    ②
            this[0] = selector;
            this.length = 1;
            return this;
        }
        if (typeof selector == "string") {//selector为string       ③
            var match = quickExpr.exec(selector);
            if (match && (match[1] || !context)) {
                if (match[1])// 第二种情况处理$(html) -> $(array)   ④
                    selector = jQuery.clean([match[1]], context);
                else {// 第三种情况：HANDLE: $("#id")//处理$("#id")
                    var elem = document.getElementById(match[3]);
                    if (elem) {
                     // IE会返回name=id的元素 ，如果是这样，就document.find(s)
                       if (elem.id != match[3])                     ⑤
                            return jQuery().find(selector);
                       // 构建一个新的jQuery(elem)
                        return jQuery(elem);                        ⑥
                    }
                    selector = [];
                }
            } else
            // 第四种情况：处理$(expr, [context])==$(content).find(expr)
                return jQuery(context).find(selector);              ⑦
        } else if (jQuery.isFunction(selector))                     ⑧
```

```
    // 第五种情况：处理$(function)七Shortcut for document ready
  return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector);
// 第六种情况：处理$(elements)
return this.setArray(jQuery.makeArray(selector));                    ⑨
    },
```

jQuery.fn.init 负责对传进来的参数进行分析然后生成 jQuery 对象。它的第一个参数一般来说是必须的（为空的话，就是默认的 document）。从源码的角度第一个参数有着如下四种**类型**：

类型	说明
Dom Element	第一个参数为 Dom 元素，第二个参数不用。直接把 Dom 元素存在新生成的 jQuery 对象的集合中。返回这个 jQuery 对象。构建 jQuery 对象完成。
String	第一个参数为 string 有三种情况： 1、html 的标签字符串，$(html) -> $(array)，第二个参数可选。执行 selector = jQuery.clean([match[1]], context);。该语句是把 hteml 的字符串转换成 dom 对象的数组。接着执行 Array 类型的返回。 2、字符串为#id 时$(id) 首先通过 var elem = document.getElementById(match[3]);取得 elem，如没有取到 selector = [];转到执行 Array 类型的返回空集合 jquery 对象。 如找到 elem,通过 return jQuery(elem);再次生成 jquery 对象，这次是 Dom Element 类型的 jquery 对象的返回。 3、兼容 css1-3 语法的 selector 字符串，第二个参数是可选的。执行 return jQuery(context).find(selector);。该语句先执行 jQuery(context)。可以看出 context 第二参数可以是任意的值，可以是集合形式。之后就通过 find(selector) 找到 jQuery(context)中所有 dom 元素都满足 selector 表达式的 dom 元素的集合，构建新的 jquery 对象，并返回。 #id 其实和这种方式是统一的，单独出来是为了提高性能。

Fn	第一参数是函数。第二个参数不用。是$(document).ready(fn)的简写，其 return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector) 是其执行的代码。这个语句首先执行 jQuery(document)，它再一次 newjQuery.fn.init 函数,生成 jQuery 对象（元素为 document）。再调用这个对象的 ready(fn)方法。Ready(fn)返回当前对象。而上面的语句又是返回这个 Ready(fn)的返回对象。 可见这个$(fn)返回是$(document)的对象。抛弃了第一次生成的$(fn)对象。
Array	第一参数是除上面提到 Dom 元素，函数，string 所有其它的类型。可以为空如$()。第二个参数不用。 语句：return this.setArray(jQuery.makeArray(selector)); 它首先是把第一个参数转换数组。Selector 可以是 Array-like 的集合，如 jQuery 对象，如 getElementsByTag 返回的 Dom 元素集合等，可能支持$(this)。Selector 还可能是单个任意的对象。 转换成标准的数组之后，执行 this.setArray 把这个数组中的元素全部存到当前 jquery 对象的集合中。之后返回当前的 jquery 对象。 其实 Dom Element 完全可能综合在这里面，单独拿起来为了提高性能。

从上面的代码和上表中，我们也可以看出构建 jquery 对象就是往 jquery 对象的集合中添加元素（一般都应该是 dom 元素）。添加的元素有两种形式：

一是单个元素，可能通过直接的 dom 元素的传参形式，还可以通过#id 从 dom 文档中找元素。

二是集合，如 jquery 对象，还有数组，还有通过 CSS Selector 找到的 Dom 集合等 Array-Like。

上表仅仅是分析传入的参数的类型，它是怎么做呢？在⑤处它实现 CSS1~CSS3 的兼容的 Selector 的查寻器的功能。通过 `jQuery().find(selector);` 来进行分析 String 并查找到符合传入的 Selector 语法的 Dom 文档树中的元素集合。

在④处，它实现了把 html 的字符串转换成 Dom 元素节点的集合。这个是通过 `jQuery.clean([match[1]], context);`来实现的。

在⑧处，它实现 DomReady 的 jQuery 对象的统一入口，我们可以通过$(fn)要注册 domReady 的监听函数。所有的调用 jQuery 实现的功能代码都应该在 domReady 之后才运行。$(fn)是所有的应用开发中的功能代码的入口。它支持任意多的$(fn)注册。其是通过 return jQuery(document)[jQuery.fn.ready ? "ready" : "load"](selector);来完成的。

找到元素之后就是构建集合了，在⑨处就是通过 `this.setArray(jQuery.makeArray(selector));`来构建 jquery 对象内部的集合。

3、构建 Jquery 的 Dom 元素

在 jQuery.fn.init 函数中，最终的结果是把 Dom 元素存放到 jQuery 对象中的集合内面。根据上一节，我们可以传入单个 Dom 元素或集合直接将其存入 jQuery 对象的集合。

如果其第一个参数是 string 类型的话如#id，那么就得到 Dom 文档树去查找。对于 html 片断的 string 类型就得生成 Dom 元素。如果再进一步分析，那些传入 Dom 元素（集）的参数从哪里来呢？它们可以通过 Dom 元素的直接或间接引用方式得到。

这一部分首先分析如何从 html 的片断就得生成 Dom 元素,然后分析 jQuery 是如何通过直接或间接的方式在在 Dom 树中找到 dom 元素，第三就是分析基于 CSS1~CSS3 的 CSS selector。

3.1 生成 Dom 元素

jQuery.fn.init 函数中通过 jQuery.clean([match[1]], context);来实现把 html 片断转换成 Dom 元素，这是一个静态方法：

```
 // 把html转换成Dom元素,elems多个html string 的数组
clean : function(elems, context) {
     var ret = [];
     context = context || document;//默认的上下文是document
     //在IE中!context.createElement行不通，因为它返回对象类型
     if (typeof context.createElement == 'undefined')
          //这里支持context为jQuery对象，取第一个元素。
       context = context.ownerDocument || context[0]
```

```
                && context[0].ownerDocument || document;

jQuery.each(elems, function(i, elem) {
     // 把int 转换成string的最高效的方法
    if (typeof elem == 'number')elem += '';
    if (!elem) return;// 为''，undefined,false等时返回
    if (typeof elem == "string") {// 转换html为Dom元素
    // 修正 "XHTML"-style 标签，对于如<div/>的形式修改为<div></div>
    //但是对于(abbr|br|col|img|input|link|meta|param|hr|area|embed)
    //不修改 。front=(<(\w+)[^>]*?)
     elem = elem.replace(/(<(\w+)[^>]*?)\/>/g,
        function(all, front, tag) { return
        tag.match(/^(abbr|br|col|img|input|link|meta|param|hr|area
        |embed)$/i)? all: front + "></" + tag+ ">";}  );
    // 去空格，否则indexof可能会出不能正常工作
    var tags = jQuery.trim(elem).toLowerCase(),
    div = context.createElement("div");//在上下文中创建了一个元素<div>
    // 有些标签必须是有一些约束的，比如<option>必须在<select></select>中间
    // 下面的代码在大部分是对<table>中子元素进行修正。数组中第一个元素为深度
    var wrap =
     //<opt在开始的位置上（index=0)就返回&&后面的数组，这是对<option>的约束
      !tags.indexOf("<opt")&& [1, "<select
                                multiple='multiple'>","</select>"]
//<leg 必须在<fieldset>内部
|| !tags.indexOf("<leg")&& [1, "<fieldset>", "</fieldset>"]
//thead|tbody|tfoot|colg|cap必须在<table>内部
||  tags.match(/^<(thead|tbody|tfoot|colg|cap)/)
                          && [1, "<table>", "</table>"]
 //<tr在<tbody>中间
|| !tags.indexOf("<tr")&& [2, "<table><tbody>", "</tbody></table>"]
 //td在tr中间
 (!tags.indexOf("<td") || !tags.indexOf("<th"))&& [3,
                "<table><tbody><tr>","</tr></tbody></table>"]
 //col在<colgroup>中间
|| !tags.indexOf("<col")&& [2,
          "<table><tbody></tbody><colgroup>","</colgroup></table>"]
 //IE中 link script不能串行化 ？
|| jQuery.browser.msie&& [1, "div<div>", "</div>"]
 //默认不修正
|| [0, "", ""];

// 包裹html之后，采用innerHTML转换成Dom
div.innerHTML = wrap[1] + elem + wrap[2];
```

```
    while (wrap[0]--)
    // 转到正确的深度,对于[1, "<table>","</table>"]，div=<table>
        div = div.lastChild;

    // fragments去掉IE对<table>自动插入的<tbody>
    if (jQuery.browser.msie) {
   // 第一种情况：tags以<table>开头但没有<tbody>。在IE中生成的元素中可能会自动
   // 加的<tbody> 第二种情况：thead|tbody|tfoot|colg|cap为tags，
  // 那wrap[1] == "<table>" .tbody不一定是tbody,也有可能是thead等等
    var tbody = !tags.indexOf("<table")&& tags.indexOf("<tbody") < 0
                ? div.firstChild&& div.firstChild.childNodes
                : wrap[1] == "<table>"&& tags.indexOf("<tbody") < 0
                    ? div.childNodes: [];
    // 除去<tbody>
     for (var j = tbody.length - 1;j >= 0; --j)
        if (jQuery.nodeName(tbody[j],
                  "tbody")&&!tbody[j].childNodes.length)
           tbody[j].parentNode.removeChild(tbody[j]);

    //使用innerHTML,IE会去开头的空格节点的,加上去掉的空格节点
      if (/^\s/.test(elem))
      div.insertBefore(context.createTextNode
                (elem.match(/^\s*/)[0]),div.firstChild);
    }

    elem = jQuery.makeArray(div.childNodes);//elem从字符转换成了数组
   }
     //采用===0,因为form,select都有length属性。这里主要是为了form,select进
     //行下面的if else 处理。对于其它的length === 0的,也根本就不要加入到ret中。
if (elem.length === 0&& (!jQuery.nodeName(elem, "form")
                 && !jQuery.nodeName(elem, "select")))
                     return;
   //不是（类）数组的形式的元素，或是form元素或是select元素（这两个可以看作类数组）
if (elem[0] == undefined|| jQuery.nodeName(elem, "form")|| elem.options)
        ret.push(elem);
else// 对于elems是array-like的集合
    ret = jQuery.merge(ret, elem);
  });
   //上面的each中把有效的元素都加入到ret，现在只要返回就得到转换的Dom元素数组
return ret;

},
```

在上面的代码中，我们可以看出对于 elems, context 的参数的支持是多种形式的，elems 可以为（类）数组的形式，还可以采用对象的形式。数组中的元素

或对象的属性可以是混合形的，如 string,ojbect,甚至（类）数组的形式。对于数字类型，会转换在 string 形，除 string 形之外的都放入返回的数组中，当然对于集合的形式，那就会取集合中每个元素。

对于 string 的形式就转换成 Dom 元素的形式，之后存到返回的数组中。这是这个函数的主要任务。如何把 html 转换成 Dom 元素，在这里是采用 innerHTML 把 html 挂到 Dom 文档树中。这样就转换成了 Dom 元素。

有些 html 标签片断是有约束的,比如<td>xx</td>，它必须存在 table 的 tr 中，也就是说在要进行 html 的标签片断的修正。这也是上面的代码处理的重点。

3.2 间接引用 Dom 节点

在jQuery.fn.init函数参数中，我们可以直接输入Dom元素（集）的参数来构建jQuery对象，那么这些Dom元素（集）从那里来的呢？可以通过直接引用方式如：document.getElementById，或getElementsByTag来引用Dom元素节点。我们还可以通过dom元素的间接引用的方式如：childNodes firstChild、lastChild、nextSibling、parentNode、previousSibling等引用Dom元素节点。
lastChild、parentNode等这些都是Dom元素的属性，jquery对象是Dom元素的集合。故它也应该把这些间接引用的属性整合到jquery对象中，这样就可以通过jquery对象来获得其集合中所有元素的各自的间接引用节点。把这些间接引用的节点组合起来构成新的集合。集合中元素是jquery对象所有元素的子节点或父节点之类的间接引用节点。

有了这些节点集合，就可以作为 jQuery.fn.init 的参数传入，这是全部操作都统一到 jQuery 类库中的使用。有人会问连这 jQuery.fn.init 都没有完成，怎么可能通过 jquery 对象而得到间接引用节点（集）呢？

第一、因为 jQuery.fn.init 的参数可以有四个类型，我们完全可以通过 String 类型来先构建 Jquery 对象(html 生成 Dom 对象，#id 的 getElementById 直接引用，CSS Selector)。然后就可以把这个对象的间接引用到的元素节点作为参数传给下一个要构建的 jquery 对象。很多调用 pushStack 函数的用法就是这样的。

第二、如果硬是想不脱离 jQuery 类库来进行间接引用的话，jQuery 也提供几个静态方法：jQuery.dir、jQuery.nth、jQuery.sibling 来完成间接引用。既然是间接引用，当然得先有一个 dom 元素做为主体。直接引用也有主体，但是我们可以采用 document 来做为默认的。因为直接引用是查找的方式。

```
jQuery.each( {// 一组对元素的相关节点的操作，如父，子，兄节点等
      parent : function(elem) {// 父亲节点
          return elem.parentNode;
       },
```

```
        parents : function(elem) {// elem的所有parentNode
            return jQuery.dir(elem, "parentNode");
        },
        next : function(elem) {// 元素的下一个兄弟
            return jQuery.nth(elem, 2, "nextSibling");
        },
        prev : function(elem) {// 前一个兄弟
            return jQuery.nth(elem, 2, "previousSibling");
        },
        nextAll : function(elem) {// 所有后继兄弟
            return jQuery.dir(elem, "nextSibling");
        },
        prevAll : function(elem) {// 所有前继兄弟
            return jQuery.dir(elem, "previousSibling");
        },
        siblings : function(elem) {// 所有兄弟
            return jQuery.sibling(elem.parentNode.firstChild, elem);
        },
        children : function(elem) {// 所有孩子
            return jQuery.sibling(elem.firstChild);
        },.. .. .. },
function(name, fn) {// 注册到jQuery对象中去，可以调用同名方法
            jQuery.fn[name] = function(selector) {
                var ret = jQuery.map(this, fn);// 每个元素都执行同名方法
                if (selector && typeof selector == "string")
                    ret = jQuery.multiFilter(selector, ret);// 过滤元素集
                return this.pushStack(jQuery.unique(ret));// 构建jQuery对象
                };
        });
```

这段代码就是 jQuery 对象的间接引用节点的方法。它提供了父节点、子节点，兄弟节点三个方法的操作。对于父节点，可以可得到其当前的父亲节点，还可以得到所有父亲节点(包括祖先节点)。对于子节点，就是所有的直接的子节点（不包括其它的后代节点）。对于兄弟节点，有大（前面）兄弟节点（们）和小（后面）兄弟节点（们）之分。

接下就是把这八个节点注册到 jQuery.fn（即 jQuery 对象）中的同名方法中。因为 jQuery 对象的 Dom 元素是集合，那么就得对集合中每个 Dom 元素都要进行单个元素相同的操作（上面 8 种的间接引用节点的 Fn）。在组合中，有可能会有重复的 dom 元素，也还有可能用户会对这些间接引用节点进行其它的自定义的过滤。这里支持的过滤方式下一小节要分析的 selector 的过滤。最后构建一个新的 jQuery 对象。

其调用的方式如：$(‘p’).parents()。这个例子是所有 p 标签的所有父辈（包

含祖先）节点的所组合而成的集合。这个集合有重复的元素可能性是很大。所以要 jQuery.unique(ret)来进行过滤。

再举一个例子：$(‘p’). parents(“div”)。这个就是取所有所有父辈（包含祖先）节点中的 Tagname 为 div 的元素集合。这个就采用了 CSS 的过滤。最后还是要进行 unique 处理。

在上面的代码，我们也看到了几个静态间接引用节点的函数调用：

```
//从一个元素出发，取得一个方向上的所有元素，如parentNode、nextSibling、
//previousSibling、lastChild,firstChild这样的元素的属性作为方向（dir）.
dir : function(elem, dir) {
    var matched = [], cur = elem[dir];
    while (cur && cur != document) {
        if (cur.nodeType == 1)
            matched.push(cur);
        cur = cur[dir];
    }
    return matched;
},
```

取名 dir，我们会想到 Dos 的 Dir 操作。其实我们解释为 direction(方向)更好理解。它一直会朝着方向走到尽头。如 parentNode 的方向。它会把取的父节点作为当前节点再取其父节点。就是所有的父辈节点（们）。对于 nextSibling、previousSibling、lastChild,firstChild 都一样。只要取到的元素有 dir 这个方向（属性）的话，就一直取下去。每一步都会把取到的元素保存起来。这个函数是对于要取‘一条线’形式的 Dom 文档树的所有元素是有很有用的。从一个元素出发，其 parentNode 可以画一条线，lastChild 也可以画一条线。当然 nextSibling 不也类外。

Dir 是取某一个方向上所有的元素，有的时候，我们要取的是某一个方向上的第几个元素，如我们找到其爷爷节点。那就是某一个方向上的第二个元素。下面的 nth 就是实现这个功能。

```
// 取某个方向上的第几个元素。Result是第几个
nth : function(cur, result, dir, elem) {
    result = result || 1;
    var num = 0;
    for (;cur; cur = cur[dir])
        if (cur.nodeType == 1 && ++num == result)
            break;
    return cur;
},
```

Nth 与 dir 在一个方向有点不同的地方，就是 dir 不包含自身，而 nth 包含自身，如果 result 是 1 的话，那是自身的元素。如果没有找到就返回空（有可能是

undifined,或 null）。

接下来的 sibling 就比较简单，但是用途也会少很多。它实现从一个元素(包括自身)找到其所有后续兄弟（们），然后从这个后续的兄弟（们）排除一个指的元素。

```
// 从包含n元素开始的所有后续兄弟，但不包含elem。
sibling : function(n, elem) {
    var r = [];
    for (;n; n = n.nextSibling) {
        if (n.nodeType == 1 && n != elem)
            r.push(n);
    }
    return r;
    }
```

仔细想一下，没有什么好适用的场合，但是 jquery.fn.siblings 中的用法倒是最合适的用法。找到自己的所有兄弟。找自己的所有兄弟最好的方法当然是找到父亲节点，然后排除自身节点。采用 childNodes 再查找当前元素并去掉和这个操作差不多，效率应该会高点。

3.3 采用 CSS 方式查找 Dom 节点

在 jquery.fn.init()中，我们对这句 jQuery(context).find(selector)没有深入去分析，在使用$()时候，大部分时间都是在使用这句来完成功能的。它就是调用 CSS Selector 到 Dom 树去查找和相符 CSS 语法的元素节点（集）。jQuery 名字中 query 的意义就体现在这里。

根据符合 CSS 语法的字符串，它是怎么到 DOM 文档树去找到符合条件的元素呢？无论怎么解析这个字符串，它总得有调用最原始的函数来完成功能，这些函数是什么？

在没有分析Selector源码之前，说来也不会相信功能强大的selector是建立在元素的getElementsByTagName，getElementById直接引用和元素的childNodes firstChild、lastChild、nextSibling、parentNode、previousSibling等间接引用这些函数的使用。

在分析源码之前，我们得了解一下 CSS selector，它可以粗略分成几类基本的类型：ID 选择器（#id），Class 选择器(.class)，类型（type）选择器(p)，Combinators，属性（Attribute）选择器，Pseudo Classes 选择器等。这些都是单一的选择器，可以在应用中把它们组合起来，如：div#id, div:last-child。

我们先分析一个例子：span[class=’test’]，它是属性选择器，我们一般把它做一个整体来理解：在 Dom 文档树中找到其 class 等于’test’的且标签为 span 的元素。这是一步到位，直接从 dom 文档树查找（select）所需要的元素。

其实我们更细一点分析，完成可以把这个字符串拆分成两步走，第一步是根据 document.getElementsByTagName(tagName)取得 span 的元素集，第二步是再根据其 class 是否等于 test 来进行判断，把不等于的元素从结果集中去掉。

不光是属性选择器，对所有的复合的选择器，都可以根据这种细粒度来拆分选择器，接下来就是分别是每个小部分进行操作。如果这样，我们就能把 CSS selector 分成两大类，第一类是是选择（selector），从根据给定的 selector 从 Dom 树找到相关的元素节点放到结果集中来。第二类是筛选(filter)。在结果集中判断该元素还满足表达式。

这样一来，就完全可以 JS 的原有直接或间接对 Dom 元素进行引用的方法，如 div#id 就可以变成先找到 div 的元素，之后就判断其元素的 id 是否等于 id，如果不等于就从结果集中删除掉。而对于 div[id=’bar’]也是一样，可以看出 div#id 与 div[id=’id’]是一样的操作。对于 div.class 也可以转换到属性中去操作。

现在要做的事就分析那些基本选择器是完成选择（selector）还是筛选(filter)的工作。我们把类型选择器，Combinators 统一称为元素选择器，其形式如下：*，E F，E~F，E+F，E>F，E/F，E。这些一定是从 dom 树中选择元素。

对于 ID 选择器和 Class 选择器，它们也可以在 selector 字符串的起始位置，说明他们也可以完成选择（selector）功能。当它们不在起始位置上，如 div#id，div.class 那他们就是筛选器。我们可以变通一下统一起来，对于 ID 选择器可以采用 document.getElementById 来直接引用。但是对于.class 就不一样了，它没有相对的函数。这种情况下，我们可以把它变成*.class。*能取得一个范围内的所有的元素，然后再进行筛选。这样对于 ID 选择器，它有专门的函数来处理，是选择（selector）器。对 Class 选择器，它是筛选器，特殊情况就是对于*标签取到元素集进行筛选。

其它如属性（Attribute）选择器，Pseudo Classes 都是筛选器，只能附在一个元素或多个元素后面。，特殊情况就是对于*标签取到元素集进行筛选。

看一个：div.foo:nth-child(odd)[@foo=bar]#aId > a 的例子：

→第一步：找到 div type 选择器的所有元素。

→第二步：在这些元素中找到 class 为 foo 的元素。如果没有就根本不要去分析下面的字符。

→第三步：根据:nth-child(odd)找到元素的子孩子元素为偶数的的元素。进一步筛选了。

→第四步：找到[],就是属性筛选器了，根据 foo 属性值 bar 来判断集合中还有那些满足条件。

→第五步：在这些元素集合，找到 id=aId 的元素，进一步筛选。

→第六步：在这些元素集合中找到所有元素对应的子元素类型为 a 的所有子元素。

→结果：我们可以看到，现在元素是子元素的集合。

讲完了通版的大道理，接下来就是来看看 jQuery 是怎么实现的：

```
// 搜索所有与指定表达式匹配的元素。这个函数是找出正在处理的元素的后代元素的好方法。
// 所有搜索都依靠jQuery表达式来完成。这个表达式可以使用CSS1-3的选择器语法来写。
    find : function(selector) {
    var elems = jQuery.map(this, function(elem) {// 找到每个元素的满足的
              return jQuery.find(selector, elem);    });
        return this.pushStack(/[^+>] [^+>]/.test(selector) ? jQuery
              .unique(elems) : elems);// 是不是返回不重复的元素？
    },
```

这是 jquery.fn.init 调用的 find 方法。它只是完成之本本对象集合中每一个元素都调用 jQuery.find(selector, elem)方法，组合成新 unique 集合后构建新 jquery 对象。最重要的实质性的工作在 jQuery.find(selector, elem)完成：

```
find : function(t, context) {
  if (typeof t != "string")return [t];// 快速处理非字符表达式
  if (context && context.nodeType != 1 && context.nodeType != 9)
        return [];// 确保context是DOM元素或document
    context = context || document;// 缺省的context
  // 初始化，ret:result, done:已经完成，last:上一次的t,nodeName:节点名，
  var ret = [context], done = [], last, nodeName;

    //这里就是把复合选择器的字符串从左到右取最小单元的选择进行分析操作
    //分析操作完之后就这个分析过的字符串部分给删除，
    //然后循环分析接下来的剩余的部分。直到字符串为空。
    //这里的最小单元指如#id,~F(+F,>F),.class,[id='xx'],F,:last()之类
 while (t && last != t) {// t存在，且变化
        var r = []; // ret的tempValue
        last = t; // last:上一次的t
        t = jQuery.trim(t);// 去首尾空格

        var foundToken = false, re = quickChild, // 以>开头的regexp
        m = re.exec(t);

      //这一部分处理了>,+,~的元素选择器。当然有的后代，有的兄弟选择的。
```

```
 // 首先判断是不是以>开头，因为每次处理都处理都删除分析过的字符串部分
 //这里可以看作是>作为找到tagName元素的子节点们的标记
if (m) {                                                    ①
        nodeName = m[1].toUpperCase();//tagName

//在结果集中（第一次是默认是给定的context）找到满足的tagName元素的所有子节点。
//两个循环，第一是对结果集中每个元素进行，第二个是对每个元素中每个子元素节点。
//找到结果集中所有的元素的所有子元素的集合。
    for (var i = 0;ret[i]; i++)
        for (var c = ret[i].firstChild;c; c = c.nextSibling)
        if (c.nodeType == 1&& (nodeName == "*" ||
                    c.nodeName.toUpperCase() == nodeName))
                r.push(c);
        ret = r; // 现在找到的所有元素都是结果集
        t = t.replace(re, "");// remove已经处理过的字符串部分

    //对于E （F，>F,+F etc)的形式，这里跳过后面的代码又回到while处执行。
    //但是在while处之后会把这个空格去掉。好像没有进行操作。这里变化了是ret。
    //无论后面是怎样的最小单元选择器，都是要根据这个个ret中的元素来进行操作。
    //如果是tagName，那么就是转4处执行ret[i].getElementsByTagName().
    //如果是>tagName,就执行1处的代码，其它的省略，
    //可见每个最小单元之后都可以是任意的空格分隔。
    if (t.indexOf(" ") == 0)continue;
    foundToken = true;// 找到标识
    }
else {// 第二判断是不是以+~开头                                ②
    re = /^([>+~])\s*(\w*)/i;
    if ((m = re.exec(t)) != null) {// 以+~开头的
        r = [];
        var merge = {};
        nodeName = m[2].toUpperCase();// 节点名
        m = m[1];// 符号，如+，~
      // 如果selector字符串的匹配+或~（子元素），
      //在结果集中所有元素中找到其兄弟元素节点。
     //不同的+找的是后续的第一个兄弟元素，
     //而~是找到所有的后续的兄弟元素节点。
     //之后把找到的所有的元素放到结果集合中。
   for (var j = 0, rl = ret.length;j < rl; j++) {
                // 把~和+的操作统一在一起进行处理
        var n = (m == "~" || m == "+"
                    ? ret[j].nextSibling: ret[j].firstChild);
     for (;n; n = n.nextSibling)
        if (n.nodeType == 1) {// 保证节点是元素类型
            var id = jQuery.data(n);// 为n元素生成全局唯一的id
```

```
                if (m == "~" && merge[id])// 保证ret中元素不重复
                    break;// nextSibling会循环到第一个节点？
                if (!nodeName|| n.nodeName.toUpperCase() == nodeName) {
                    if (m == "~") merge[id] = true;
                r.push(n);
                }// 直接后续兄弟节点，只进行一次操作。
            if (m == "+") break;
          }
        }
    ret = r;// 找到的所有的元素放到结果集合中。
    t = jQuery.trim(t.replace(re, ""));
    foundToken = true;
    }
}

// 不是以>~+开头的或者说除去已经分析过的字符，接下来的字符是不是>~+开头
  if (t && !foundToken) {                                                    ③
    //这里的意思是在开始的位置找到,号，说明一个selector已经完成了，那么
    //结果集就要存到已经完成的集合中。结果集也应该初如化。
    if (!t.indexOf(",")) {                                                   ④
        //说明运行到这里的时候，还是单个selector的字符串分析是刚刚开始
         //因为>~+不可能得到ret[0]元素等于元素的自身。如果等于的话，
        //那就清除出ret,因为接下来就要把ret结果集中的元素存入done中
        if (context == ret[0])  ret.shift();
        done = jQuery.merge(done, ret);// ret的其它元素放入done
        r = ret = [context];// 重新初始化
        t = " " + t.substr(1, t.length); //把,采用空格代替。
            }
   else {                                                                  ⑤
        // 说明这一个selector部分还没有完成，同时还没有找到元素
        // 或者是 >F的后面用空格来分隔。
         //* qId:^((?:[\w*_-]|\.)+)(#)((?:[\w*_-]|\.)+)
        // * qclass:^([#.]?)((?:[\\w*_-]|\.)*)
        var re2 = quickID;// 如(.)nodeName#idName
        var m = re2.exec(t);// 找到第一个相配的

    if (m) {m = [0, m[2], m[3], m[1]];// m=[0,#,idName,nodeName]}
    else { re2 = quickClass;// #nodeName,.className
           m = re2.exec(t);// m=[all,#,idName]}

    m[2] = m[2].replace(/\\/g, "");// 去除转义字符

    //取数组的最后一个元素，其实就是为了取到是不是为document,
```

```
    //因为只有document才有getElementById，为什么不直接采用
    //document呢？难道document的查找会比先根据element.
    //getElementsByTagName找到元素下面的tagname的相符的
    //集合然后采用id属性去判断对比来得更慢吗？不一定？对于大的Dom树，
    //而element的范围又很小的话，可能会慢一些。
    //不过由于这里还要通过属性选择器来进行验证进行验证，一般来说
    //element.getElementsByTagName会快一点。
    var elem = ret[ret.length - 1];
    if (m[1] == "#" && elem && elem.getElementById
                    && !jQuery.isXMLDoc(elem)) {
           var oid = elem.getElementById(m[2]);

        // 回测元素的ID的确存在，在IE中会取name属性的值，同时在form 元素中
        // 会选中在form中元素的name属性为id的元素。
        if ((jQuery.browser.msie || jQuery.browser.opera) && oid
                    && typeof oid.id == "string" && oid.id != m[2])
               //通过属性选择器来进行验证。
            oid = jQuery('[@id="' + m[2] + '"]', elem)[0];

            // 回测元素的node Name是否相同，如div#foo,可以提交效率
            ret = r = oid && (!m[3] || jQuery.nodeName(oid, m[3]))
                        ? [oid]: [];
               }
         else {
            //这里处理了#id,.class tagName,div#id四种形式
            //这里我们还可以看出E F形式。并没有特殊的处理。就是采用了
            //E.getElementsByTagName(F)就可以了。
            //这个就能取后元素的所有后代为F和F的元素节点。
            //和F使用是统一的起来。因为E都是结果集。
         for (var i = 0;ret[i]; i++) {
//因为m有两种情况：[0,#,idName,nodeName]、[all,#/.,idName/class/tagName]
//这里就是根据这两种情况来取得tagName，m[1] == "#" && m[3]
//为真，说明是第一种，取nodeName。如果m[1] == "#" && m[3]为假
//说明m[1] <> "#"||!m[3],而m[1] != ""说明只能是第二个数组中的.或#
//说明了对于#nodeName,.className采用element.getElementsByTagName(*).
//当m[1] == ""，说明是一个E 元素选择器，它不带任何的前缀。如：p。这个时候
//m[2]是tagName.
//m[0] == "" ，只能指第二个数组中的。它==",说明没有找到符合qclass的regExp.
//其它的情况都不会为",它为",!m[1],!m[2]也都为true.
        var tag = (m[1] == "#" && m[3] ? m[3] : (m[1] != ""
                        || m[0] == "" ? "*" : m[2]));// 分情况取tagName

        //*的情况下，对于object标签转换为param标签进行操作。
    if (tag == "*"&& ret[i].nodeName.toLowerCase() == "object")
```

```
            tag = "param";// Handle IE7 being really dumb about <object>s
        //把结果集合中第一个元素的getElementsByTagName存入到临时的结果集中。
            r = jQuery.merge(r, ret[i].getElementsByTagName(tag));
        }

    //class选择器的话，就根据class属性在找到结果集合中过滤
        if (m[1] == ".")  r = jQuery.classFilter(r, m[2]);        ⑦
         //id选择器的话，就根据id属性在找到结果集合中过滤
            if (m[1] == "#") {
                var tmp = [];
                for (var i = 0;r[i]; i++)
                if (r[i].getAttribute("id") == m[2]) {
                        tmp = [r[i]];
                        break;
                        }
                    r = tmp;
                }
            ret = r;
            }
     t = t.replace(re2, "");
    }

    }

        //这个时候已经找到结果的集合，对于如CSS Selector为:hidden的属性筛选器，
        //它的集合就是context的下面的所有元素节点。也就是说上面的
        //代码无论如何都能找到元素的集合。这个集合可能是>/+~ F
        //或#id,.class tagName,div#id，对于不满足这些条件的，就采用
        //context.getElementsByTagName(*)要取得其下所有的元素
        //确保接下来的过滤（筛选）
if (t) {// 根据余下的selector，对找到的r集合中的元素进行过滤      ⑥
            var val = jQuery.filter(t, r);                        ⑧
            ret = r = val.r;
            t = jQuery.trim(val.t);// 去首尾空格
        }
    }

    //如果还会有t存在说明一个问题：last == t
    //也就是上一次的过程中没有进行成功的解析一个最小单元的选择器
    //原因是输入的 t 字符串有语法上的错误。如果是采用,分隔的多选择器
    //那么就是当前及之后的选择器不分析。完成的done就是之前的结果集。
    //觉得这样处理不好，很多时间我们都会写错CSS selectror,不报错，
    //对于调试发现问题特难。
    if (t) ret = [];
```

```
    //出现这种情况说明运行到这里的时候，还是单个selector的字符串分析是刚刚开始
    //如果等于的话，那就清除出ret,因为接下来就要把ret结果集中的元素存入done中
    if (ret && context == ret[0])
        ret.shift();// 去掉根上下文

    done = jQuery.merge(done, ret);// 合并

    return done;
},
```

上面 find 的实现的代码有点长。其实分析起来也不是很难。①②③④⑤处两个 if-else 是元素选择器，针对>/+~ F 或#id,.class tagName,div#id 这样的选择器进行查找元素，构成结果集。⑥实现的就是分析它之后的属性选择器进行筛选。这段代码说白就是 select 与 Filte 之间的交互分析 CSS selector 的字符串进行查找或筛选的过程。

具体的细节在代码中有分析。在⑦处它采用先找到所有元素然后对每个元素进行 class 的判断来分析如.class 这样 selector。因为它在起始位说明是查找器。但是这里对这个最小的单元部分，我们还可能两个部分，找到所有元素，然后一个个排查，排查就是采用了 jQuery.classFilter(r, m[2])：

```
classFilter : function(r, m, not) {
    m = " " + m + " ";//这里的做法就是怕出现如"class12"这样的问题
    var tmp = [];      //如果传的m= class1,这个的条件是满足的，实际是不。
    for (var i = 0;r[i]; i++) {
        var pass = (" " + r[i].className + " ").indexOf(m) >= 0;
        if (!not && pass || not && !pass)
            tmp.push(r[i]);
    }
    return tmp;
    },
```

jQuery.classFilter 是比较简单。在 find（）的第二个部分是筛选，它调用 jQuery.filter(t, r)来完成功能：

```
//根据CSS selector表达式查找集合中满足该表达式的所有元素
//还可以根据not来指定不满足CSS selector表达式元素集
filter : function(t, r, not) {
    var last;
    while (t && t != last) {// t存在，且改变
        last = t;
        // Match: [@value='test'], [@foo]
        // 1、^(\[) *@?([\w:-]+) *([!*$^~=]*) *('?"?)(.*?)\4 *\]/,
```

```
        // Match: :contains('foo')
        // 2、^(:)([\w-]+)\("?'?(.*?(\(.*?\))?[^(]*?)"?'?\)/,

        // Match: :even, :last-child, #id, .class
        // 3、new RegExp("^([:.#]*)(" + chars + "+)")],

        //这里可以看出我们直接调用filter的时候的selector如不是筛选器的话，
        //那就不进行筛选了，这里的selector语法如[@value='test'], [@foo]、
        //:contains('foo')，:even, :last-child, #id, .class的形式
        //可以是上面这几种形式的组合，但不能包括元素选择器。
    //而且复合的形式中间不能采用空格隔开，如[@value='test']#id.class可行的。
        var p = jQuery.parse, m;
        for (var i = 0;p[i]; i++) {// 找到与jQuery.parse中regexp相配的
            m = p[i].exec(t);
            if (m) {
                t = t.substring(m[0].length);//删除处理过的字符部分
                m[2] = m[2].replace(/\\/g, "");// 有可能会有没有转换的\去掉
                break;
            }
        }
        // 与上面三种的regexp都不相配
        if (!m)    break;

        //处理 : not(.class)的形式，返回集合中不包含.class的其它的元素
        if (m[1] == ":" && m[2] == "not")
            // 性能上优化 m[3]是.class经常出现
            r = isSimple.test(m[3])// isSimple = /^.[^:#\[\.]*$/
                    ? jQuery.filter(m[3], r, true).r: jQuery(r).not(m[3]);

        //处理.class的过滤。只要看看m[2]这个class是不是被集合中元素的class包含。
        else if (m[1] == ".")// 性能上优化考虑
            r = jQuery.classFilter(r, m[2], not);
            //处理属性过滤。如[@value='test']形式的属性选择
        else if (m[1] == "[") {
            var tmp = [], type = m[3];// 符号，如=

            for (var i = 0, rl = r.length;i < rl; i++) {
                //jQuery.props[m[2]]进行tag中属性名和对应的元素的属性名转换，
                //因为tag中属性名是元素中简写，z取到 元素的属性值
                var a = r[i], z = a[jQuery.props[m[2]] || m[2]];

                //直接取元素的属性值，没有取到，说明有的浏览器不支持这种方法
                //进一步尝试采用jQuery.attr来进行非标准的兼容取属性值。
                //就算是取到了值，但对于属性名为style|href|src|selected，
```

```
            //它们不能直接取值，要进行特殊的处理，这个在jQuery.attr进行。
            //其实这里可以直接采用jQuery.attr(a, m[2])，一步到位。
            if (z == null || /style|href|src|selected/.test(m[2]))
                z = jQuery.attr(a, m[2]) || '';// 几个特殊的处理

        //如果属性选择器满足这
          //[foo],[foo=aa][foo!=aa][foo^=aa][foo$=aa][foo~=aa]
            //这几种方式之一，这个元素就可能通过。即满足条件。m[5]属性值。
    if (   (type == "" && !!z//[foo]
       || type == "=" && z == m[5]//[foo=aa]
       || type == "!=" && z != m[5]//[foo!=aa]
       || type == "^=" && z&& !z.indexOf(m[5])//[foo^=aa]
       || type == "$="    && z.substr(z.length - m[5].length) == m[5]
       || (type == "*=" || type == "~=")&& z.indexOf(m[5]) >= 0
         )
    ^ not)
        tmp.push(a);
        }
        r = tmp;

    }
    //处理：nth-child（n+1）。其实这里也改变了结果集，
    //不过这里是采用的是间接引用的方式，只要知道元素就可以了，
    //不需要dom树去查找。因为它要解析参数中的表达式
    else if (m[1] == ":" && m[2] == "nth-child") {// 性能考量
        var merge = {}, tmp = [],
    // 分析：nth-child（n+1）中的参数，这里支持
    //'even', 'odd', '5', '2n', '3n+2', '4n-1', '-n+6'几种形式
    //test[1]="-或空"，test[2]="n前面的数或空"，test[3]="n后面的数或空"
    //这样把参数分成三个部分：1是负号的处理，2是xn中的x处理，3是n-x中-x的处理
        //3中的是带有符号的。也就是+或-。
        test = /(-?)(\d*)n((?:\+|-)?\d*)/.exec(m[3] == "even" && "2n"
                || m[3] == "odd" && "2n+1" || !/\D/.test(m[3]) && "0n+"
                + m[3] || m[3]),

        // 计算(first)n+(last)
        first = (test[1] + (test[2] || 1)) - 0, last = test[3] - 0;

        //找到符合(first)n+(last)表达式的所有子元素
    for (var i = 0, rl = r.length;i < rl; i++) {
        var node = r[i], parentNode = node.parentNode,
    id = jQuery.data(parentNode);//为该元素parentNode分配了一个全局的id

        if (!merge[id]) {// 为元素的每个子节点标上顺序号，作了不重复标识
```

```
                var c = 1;
            for (var n = parentNode.firstChild;n; n = n.nextSibling)
                    if (n.nodeType == 1)n.nodeIndex = c++;
                merge[id] = true;
            }

            var add = false;//初始化add的标记

            //常数的形式，如1，2等等，当然还要判断元素的序号和这个数是否相等。
            if (first == 0) {// 0不能作除数
                if (node.nodeIndex == last)
                    add = true;
            }
            // 处理3n+2这种形式同时表达式要大于0
            //当前的子节点的序号要满足两个条件：
            //1、其序号进行first求余的结果=last.
            //2、其序号要大于last。对于-n的形式，要大于-last.
            else if ((node.nodeIndex - last) % first == 0
                    && (node.nodeIndex - last) / first >= 0)
                add = true;

            if (add ^ not)    tmp.push(node);
        }

        r = tmp;

    }
    else {// 根据m[1]m[2]在Query.expr找到对应的处理函数
        var fn = jQuery.expr[m[1]];
        //支持一个符号（如：last）后的方法名与函数的对应
        if (typeof fn == "object")
            fn = fn[m[2]];

            //支持更简短的string来代替jQuery.expr中的funciton。
            //这里没有用到。
        if (typeof fn == "string")
            fn = eval("false||function(a,i){return " + fn + ";}");

    // 执行处理函数fn过滤。对于r中每个元素，如果fn返回的结果为true，保留下来。
        r = jQuery.grep(r, function(elem, i) {
            return fn(elem, i, m, r);
        }, not);
    }
}
```

```
    return {
        r : r,
        t : t
    };
    },
```

jQuery.filter 完成分析属性([])、Pseudo(:)，class (.),id(#),的筛选的功能。从给定的集合中筛选出满足上面四种筛选表达式的集合。针对于 find()。这个 filter 完成不表明整个 selector 的分析完成了。还会交替地通过查找器来查找或通过该函数来筛选。对于单独使用这个函数，表达式中就不应该含有查找的 selector 表达式了。筛选是根据[、:、#、.这四个符号来做为筛选器的分隔符的。

class 筛选器是通过 classFilter 来完成。它还把 Pseudo 中：not、：nth-child 单独从 Pseudo 类的中单独提起出来处理。对于[的属性筛选器，实现起来也是很简单。除去这些，它还调用 jQuery.expr[m[1]];来处理 Pseudo 类，而 jQuery 还做了扩展。jQuery.expr 中的 Pseudo 类有以下几个部分：// Position Checks、// Child Checks、// Parent Checks、// Text Check、// Visibility、// Form attributes、// Form elements、// :has()、// :header、// :animated。也就是说在 CSS selector 中，我们可以采用 Pseudo 类提供上面这些类别的方法来进行筛选。其代码就是一些判断，不作分析。

如果想详细了解这样怎么用吧，推荐由一揪整理编辑jquery的中文文档。，

对于 CSS selector，尽管多次分析 domQuery，和 jquery Selector。尽管自己也吃透。但是感觉到写出来还是不到位。如果读者之前没有接受这方面。建议仔细分析上面的代码和注释。找一个复杂的例子，自行分析一下，或许可能弄懂 selector 的设计。

4、jQuery 类数组的分析

上一节我们就如何查找元素进行了分析。查找到了元素（集）之后，我们应该怎么办呢？这个查找返回的可能是集合也有可能是对象，肯定是要找个地方保存起来，那么存在哪里？又是如何去存呢？存完之后，又是如何去取或进行其的操作呢？

4.1、类数组构建

从上节可以看出 jquery 构建函数完成了查找或转换或其它的功能，其结果就

是查找到元素。Dom 树查找，html 转换成 Dom 元素，还是直接传入 Dom 元素都不过是方式而已。找到这些元素就得找个地方去存储起来。

存储有序数据的地方(集合)在JS中最好的当然是数组。那么又如何在jQuery内面实现数组呢？可以采用如下的方式：

jQuery.fn.prototype=new Array();

在上一节中的 `this.setArray(arr)`函数中加上

Array.apply(this,arr);

如果还要完美一点，就加上：

jQuery.fn.prototype.constructor=jQuery。

这样我们继承了数组的所有特性，又可以在 Jquery 对象进行数组的功能扩展。但是 jQuery 并没有这样采用继承 Array 而实现这个内部的集合。它采用了 Array-Like 的对象的实现（见 JavaScript: The Definitive Guide, 5th Edition7.8 节）。

类数组的对象还是对象，只不过像数组。数组与对象其实是没有什么好大的区别的，有序和无序的集合是它们区别。这个区别反应在数组有的 length 属性。当添加元素它会自动加上相对的个数，当删除元素，它会自动减去相对的个数。

看一下 jQuery 是怎么实现的：

```
// 第一种情况 Handle $(DOMElement)单个Dom 元素，忽略上下文
        if (selector.nodeType) {                                    ②
            this[0] = selector;
            this.length = 1;
            return this;
        }
```

这是它的第一种实现方法，通过 this[0]来直接设定第一个位置的 Dom 元素，同时设定 length=1。这里我们可以看出对象与数组一样都是采用 key/Value 对的形式存在对象中。上面的 Json 形式为{0：aDom,length=1}。这里细细分析一下数组，数组是继承于对象。其[]的解释分析最终结果可以看作{}构的对象，对[]或数组的构建时会进行把 index(如 0.1,….)做为对象属性的 key。把数组中的值做为其对应的 value。同时改变 length 的值。这也就是说为什么本质上对象与数组没有多大的区别。在很多的源码中，如 YUI 中都采用对象的形式来构建多维数组。

```
this.setArray(jQuery.makeArray(selector));
```

是其第二种实现的方法，上面实现是单个的元素，这个实现的多个元素的集合。它首先调用了 `jQuery.makeArray(selector)`这个静态方法把集合（类数组）转换成数组。

上面已经分析了数组和对象都可以采用 obj.[attr]的形式来取得其 key 对应的 value。对于集合或类数组，必须要求其实现 length 属性，有了 length 的长度，那么就从 0~length-1 的 key 属性中取得对应的 value 就可以了：

```
//把类数组的集合转换成数组，如果是单个元素就生成单个元素的数组。
 makeArray: function( array ) {
    var ret = [];
    if( array != null ){ var i = array.length;
    //单个元素，但window, string、 function有 'length'的属性，加其它的判断
      if( i == null || array.split || array.setInterval || array.call )
          ret[0] = array;
      else//类数组的集合
          while( i )  ret[--i] = array[i];//Clone数组
     }
   return ret;

    },
```

生成了一个标准的数组，那么接下来 setArray 来干什么呢？

```
// 把array-like对象的元素全部push当前jquery对象。
    setArray : function(elems) {
        this.length = 0;//初始化长度，因为push会在原始的length++
        Array.prototype.push.apply(this, elems);
        return this;
    },
```

这个调用了 Array.prototype.push 来自动修改 length 的属性值（当然是加入了元素）。由此可以推想到 Array 中众多的方法(如 shift)都可以看作改变 length 的值在对象的 key/value 对中完成无序到有序或重新排序的工作。实际上 Array 等是采用 C 或 C++来实现的。但是它构出的 JS 特性让我们可以这样去思考采用 JavaScript 的实现方式。

上面的 setArray(elems)函数只是会改变当前 jQuery 对象的集合，它会清除这个对象集合中以前的元素。但是有的时候我们想保存原来的集合中元素，同时也能就新传入的元素进行 jquery 对象的操作。它提供了 pushStack 函数来新建一个 jQuery 对象同时保存原来对象的引用。这样就可能在需要时用到自己所要的对象：

```
    pushStack : function(elems) {// 采用jQuery构建新对象，同时引用老对象。
        var ret = jQuery(elems);// 构建新的jquery对象
        ret.prevObject = this;// 保存老的对象的引用
        return ret;
    },
```

返回的是新构建成的对象，有着 jQuery 对象的全部功能，同时还可以通过 prevObject 来访问原来的老对象。

4.2 类数组的操作

构建了类数组，那么还得提供一些方法来操作这个集合，对于集合的操作无非就是定位元素，查找元素，复制（slice）和删除的操作（splice）等。jQuery 还提供 each,map 的扩展。

这些方法只和集合相关，与集合的元素的无关。jQuery 提供了众多的和其中元素（DOM 元素）相关的方法。

4.2.1 元素定位或取数

Get 和 index 的方法是集合操作中最基本的方法。基本每一种集合的操作都会提供。jquery 提供了两个类似的取元素的方法，get(index)和 eq（index）元素。

其不同之处在于 get 取得是集合中的元素，而 eq 则是返回该元素的 Clone，不会修改数组。我们还可以直接通过[i]来直接取元素，用来代替 get(i)。不过 get() 在没有参数时，则是获得全部元素。数组[i]的方式做不到。

```
// 取到本jquery对象的第几个Dom元素，无参数，代表全部的Dom元素
    get : function(num) {
        return num == undefined ? jQuery.makeArray(this) : this[num];
    },
// 获取第N个元素 。这个元素的位置是从0算起。
    eq : function(i) {
        return this.slice(i, +i + 1);
    },
```

对于元素的定位，jquery 提供了 index 方法：

```
  / 找到elem在本jquery对象的位置(index)
    index : function(elem) {
        var ret = -1;
        return jQuery.inArray( // 如是jQuery对象就取第一个元素
                elem && elem.jquery ? elem[0] : elem, this);
    },

     // 判断elem元素在array中的位置(index) 静态方法
    inArray : function(elem, array) {
            for (var i = 0, length = array.length;i < length; i++)
                // Use === because on IE, window == document
                if (array[i] === elem)
                    return i;
            return -1;
        },
```

inArray 是 jQuery 的静态方法，而 index 是通过调用 inArray 来实现其定位的功能。Index 的函数支持的参数可以是 jQuery 对象或 Dom 元素，而 inArray 则是实用方法，支持任何的元素。

4.2.2 元素的复制及追加

Jquery 提供了如数组中 slice 复制的功能的方法，还提供类似 concat 的静态方法 merge。Slice 是通过 Array 中 slice 来实现的：

```
  / 代理数组的slice,同样的操作。
    slice : function() {
        return this.pushStack(Array.prototype.slice.apply(this,
arguments));
    },
```

它返回生成新的 jQuery 对象。这个对象的集合就是要复制后的集合。对于 merge，它是静态方法，实现把第二个元素追加到第一个参数的数组中。

```
     // 把second 元素追加到first的数组中。
       merge : function(first, second) {
     // We have to loop this way because IE & Opera overwrite the length
     // expando of getElementsByTagName
       var i = 0, elem, pos = first.length;
       // Also, we need to make sure that the correct elements are being
       // returned (IE returns comment nodes in a '*' query)
           if (jQuery.browser.msie) {
               while (elem = second[i++])
                   if (elem.nodeType != 8)
                       first[pos++] = elem;

           } else
               while (elem = second[i++])
                   first[pos++] = elem;

           return first;
       },

// 把与表达式匹配的元素添加到jQuery对象中。array(-like)的集合也可以追加进来
    add : function(selector) {
        return this.pushStack(jQuery.unique(jQuery.merge(this.get(),
                typeof selector == 'string' ? jQuery(selector) : jQuery
                        .makeArray(selector))));
    },
// 把先前jQuery对象的所有元素加到当前的jQuery对象之中
```

```
    andSelf : function() {
        return this.add(this.prevObject);
    },
```

4.2.3 元素的过滤

我们可以把其它的元素增加到类数组中来，那么有的时候也得到把类数组中的不适合的元素剔出类数组。Jquery 对象提供了 filter,not 来过滤那样不用的元素。

```
    // 筛选出与指定表达式匹配的元素集合。可以通过函数来筛选当前jQuery对象的
    // 元素，还有通过用逗号分隔多个表达式来筛选
    filter : function(selector) {// grep，multiFilter的综合
        return this.pushStack(jQuery.isFunction(selector)
                && jQuery.grep(this, function(elem, i) {
                    return selector.call(elem, i);
                }) || jQuery.multiFilter(selector, this));
    },
```

filter 其实是 grep，multiFilter 的综合，如果参数是函数的话，就采用 jQuery.grep 来完成，否则采用 jQuery.multiFilter 来进行 selector 方式的过过滤。可以看出参数也只能是 Fn,String 这两种类型。

jQuery.grep 提供了以自定义的函数回调的形式来过集合的不需要的元素，最后形成需要的数组，和 map 函数有点类似。

```
// 过滤elems中满足callback处理的所有元素，inv是否相反
    grep : function(elems, callback, inv) {
            var ret = [];
            for (var i = 0, length = elems.length;i < length; i++)
                if (!inv != !callback(elems[i], i))
                    ret.push(elems[i]);
            return ret;
        },
```

代码中!inv != !callback(elems[i], i)这一句比较难理解。它的意义是如果显示指定 inv==true。那么 callback 就得返回 false 才是符合条件。这里采用三个！就是为了处理这种默认的情况。!inv 在 undefined,””,false,null,0 的时候都会为真，而！callback 的返回值如果也是这五种之一的情况也会为真。而它们之间有一个！=，说明不能同时为真为假时才执行 ret.push(elems[i]);

一般情况，我们不设定 inv 参数，那么只要 callback 函数的返回值不为 undefined,””,false,null,0 之一就是该元素满足条件。不会被剔除。

jQuery.multiFilter 与 jQuery.filter 的区别不大。multiFilter 支持采用,分隔的 selector 多表达式方式。

```
 multiFilter : function(expr, elems, not) {
```

```
    var old, cur = [];
    while (expr && expr != old) {// 存在且改变
        old = expr;
        var f = jQuery.filter(expr, elems, not);
        expr = f.t.replace(/^\s*,\s*/, "");
        cur = not ? elems = f.r : jQuery.merge(cur, f.r);
    }
    return cur;
    },
```

jQuery.multiFilter 是作筛选之用。和 jQuery.filter 一样，selector 的多表达式也能只是筛选器的组合。以 •，#，：，[，这四种符号做分隔的的表达式。

not 也是根据 selector 来剔除不符合条件的元素，但是 not 是建立在于 filter 的基础之上。它的效率会高于 filter。

```
    // 删除与指定表达式匹配的元素
  not : function(selector) {
    if (selector.constructor == String)// 对于string，就构建[]
     if (isSimple.test(selector))//.class1
      return this.pushStack(jQuery.multiFilter(selector, this, true));
    else// 多表达式要过滤
      selector = jQuery.multiFilter(selector, this);

  //这里是判断selector是不是集合
  var isArrayLike = selector.length
              && selector[selector.length - 1] !== undefined
              && !selector.nodeType;
  return this.filter(function() {// 过滤掉return false的元素
       return isArrayLike ?
          jQuery.inArray(this, selector) < 0  : this != selector;
              });
    },
```

分析 not 函数，我们首先从参数分析开始，它支持 string 的 CSS selector 表达式，还支持（类）数组这样的集合及 dom 元素。如果 selector 表达式是 class 单表达式筛选的话，那就调用 jQuery.multiFilter 过滤之后直接返回。这个本来是不需要做判断单独处理。但是这样会提高效率，很多时间都是采用单表达的 class 的形式来进行筛选。

它为什么可以独立返回，而可以不用和 this 中的元素进行比较？因为.class 不会改变 this 的集合的元素，只是剔除一些元素。而对于像#、：的有些操作是会完成变更返回的集合。这个集合是在 this 的集合的基础之上新构建的集合，会增加一些元素。有那么就得比较。

多表达式返回的是数组，如果传入的参数是元素或（类）数组，它们会统一

起来，然后调用 filter（function）的来实现比较。对于数组，就是取 this 的集合与 selector 的集合的交集。这样就保证了采用 CSS 筛选或传入的数组筛选之后的元素一定是 this 的集合的元素。对于传入的参数是单个的元素，那就直接比较，如果 this 中有这个元素，就返回这个元素。

可以看出 not 比 filter 的适用范围更广，而且使用 filiter 可能还会有潜在的错误，因为它的 CSS selector 的返回的元素集合很有可能有的元素不包含在当前的 this 的集合之中。这在某一些场合是很有用的。但是一般不明白原理的用户会感觉到莫名其妙的错误。在使用中最好还是采用 not，而不是 filter。

在上一节的 add 的函数中，我们看到 jQuery.unique（），看名字就知道它是对集合的元素进行唯一性处理，也就是集合的元素没有重复的。这个 unique 只是针对于 Dom 元素。如 Ext.DomQuery 中的 nodup 一样处理。

```
// 判断数组中的元素是否有重复的元素，返回不重复的所有元素。
  unique : function(array) {
    var ret = [], done = {};
      try {
         for (var i = 0, length = array.length;i < length; i++) {
                var id = jQuery.data(array[i]);
            if (!done[id]) {  done[id] = true;  ret.push(array[i]);}
         }
      } catch (e) { ret = array;}
      return ret;
  }
```

在分析 Ext.DomQuery 中的 nodup 时，很长时间都没有弄明白其意思（nodup=noduplicate）。这里 nodup 的作用是一样的。它对数组中每一个元素都扩展一个属性，设定全局唯一的 id 标识。因为数组的元素都是对 dom 元素的引用，判断是否重复，就只要判断元素的扩展的 id 标识是否重复就可以。重复就不加入结果集。Done 就是保存已经加入结果集合的所有 id。如果元素的 id 标识在这个 done 中就说明已经加入了这个元素，不需要再加。

4.2.4 元素的映射

集合中的元素的映射是功能强大且非常实用的的方法。Juery 如其它的集合一样提供了 each、map 两个映射的函数。Each 是对集合中每个元素都执行回调函数。而 map 还收集每个回调的返回结果组成一个新的集合，看起来仿佛两个集合之间有着 callback 的关系，采用 map 名字就能反映其意思。

对于 each，jquery 提供一个实例方法和静态方法，实例方法调用了 jQuery.map()来完成其 each 的功能。

```
// 当前jquery对象中每个元素都执行callback(index,elem)函数
each : function(callback, args) {// 返回this
// 其调用了jQuery的静态方法。prototype中的mothodize是解决这类问题的好方法
    return jQuery.each(this, callback, args);
},
```

它通过调用 jQuery.each 这个静态方法来完成功能的：

```
// 对object中的每个对象都执行callback函数进行处理。args仅仅内部用
each : function(object, callback, args) {
       var name, i = 0, length = object.length;
       // 和else的处理差不多，args的传参代替object的属性值
       if (args) {
           if (length == undefined) {
               for (name in object)
                   if (callback.apply(object[name], args) === false)
                       break;
           } else
               for (;i < length;)
                   if (callback.apply(object[i++], args) === false)
                       break;
       } else {
         // 不是array-like的object，对每个属性进行callback函数的调用
          if (length == undefined) {
          for (name in object)
             if (callback.call(object[name], name, object[name])
                  === false)
                 break;
             } else
              // array-like object,采用数组的形式来处理
                for (var value = object[0];i < length    &&
               callback.call(value, i, value) !== false; value =
               object[++i]) {}
       }
       return object;
   },
```

该静态方法支持第一个参数的类数组（数组）或对象。是数组就对每个元素进行 callback 的操作。如果是对象，就是对每个属性值进行 callback 的操作。这个 callback 回调函数的格式如下：callback:function(index,value)。Index 是索引号，value 是数组的 index 对应的元素或对象的第 index 个处理的属性。如果使用 args 参数，那 callback 回调函数的格式如下：callback:function(args)。Args 是给回调函数设定参数。再看一下 jQuery 对象的 each。它的第二个参数 args 就是采用传入的 args 直接进行给 callback 设定参数，而不是默认提集合中 index 和对应的元素，但执行的次数还是集合的 length 次。

而 Jquery 的 map 函数则是将一组元素转换成其他数组（通过回调函数返回值组成的）

```
// 将一组元素转换成其他数组  然后根据这个数组构建新的 jquery 对象。
map : function(callback) {
    return this.pushStack(jQuery.map(this, function(elem, i) {
        return callback.call(elem, i, elem);
    }));
},
```

这个函数首先通过 jQuery.map(this, function(elem, i){}来把 this 的 jQuery 对象集合的每个元素当作回调函数的 elem 的参数传到回调函数中。而这个回调函数又执行实例方法的 map : function(callback)中 callback 函数，也就是 jQuery.map 中的回调仅仅是传参代理的功能。jQuery.map 通过代理的回调来取得转换而成的元素集合。

接下来就是采用 pushStack 把这集合的元素构建成新的 jQuery 对象并返回，同时保存原 jQuery 对象的引用。

看下 Map 的静态方法：

```
// 返回对elems每个元素都进行操作的callback函数的返回值的集合。
    map : function(elems, callback) {
        var ret = [];
        for (var i = 0, length = elems.length;i < length; i++) {
            var value = callback(elems[i], i);
            if (value != null)//说明转换的集合的个数可能少于原来的集合
                ret[ret.length] = value;
        }
        return ret.concat.apply([], ret);//采用array的concat实现
    }
```

Map 的静态方法返回每个 callback 返回的元素组成的数组。

4.3 其它操作

对于 Jquery 对象的集合操作，还有着一些其它的方法，如判断集合的元素是否满足某一个表达式或条件，满足就返回真。在上面的集合操作中，我们经常把老的集合 pushStack。那么又如何取得到进栈的这个老集合呢？

Jquery 提供了 is 方法来完成对集合的元素判断

```
// 用一个表达式来检查当前选择的元素集合，
```

```
  //如果其中至少有一个元素符合这个给定的表达式就返回true。
 is : function(selector) {
    return !!selector && jQuery.multiFilter(selector, this).length > 0;
   },
```

!!可以把任何元素转换成 boolean 型。开发者经常使用的判断是对 class 的判断，于是提供了一个便利的操作：

```
   // 检查当前的元素是否含有某个特定的类，如果有，则返回true
   hasClass : function(selector) {
       return this.is("." + selector);

   },
```

End 就是找到上一个 jquery 对象。

```
// 回到最近的一个"破坏性"操作之前。即，将匹配的元素列表变为前一次的状态。
   end : function() {
       return this.prevObject || jQuery([]);

   },
```

第二篇 manipulate（操作）

5 DOM 元素

构建了 jquery 对象，也能对 jquery 的集合中元素进行局部的调整。现在就是操作。对于 jquery 对象中 Dom 元素进行操作。对 jquery 进行 dom 的操作就是对 jquery 对象集合的所有元素都进行操作（有的时间只是第一个元素）。

对于 dom 的操作可以分成对元素的属性，内容，CSS，insert 这几个方面进行操作。

5.1 dom 元素的属性

对 dom 元素的操作，对元素的属性进行操作是很重要的一项。我们可以通过 dom 元素的原始方法对元素元素进行操作，但是由于浏览器的兼容等各方面的问题，jquery 和其它的 lib 一样，都提供了一个完好兼容的操作。

5.1.1 Attr

名称及描述	返回	兼容性
getAttribute(name) 当前节点给定Name的属性值	Object	All
getAttributeNS(namespace, name) 当前节点给定namespace,Name的属性值	Object	All
getAttributeNode(name) 取当前节点给定name的属性节点。	Attr	All
getAttributeNodeNS(namespace, name) 取当前节点给定namespace,name的属性节点。	Attr	All
hasAttribute(name) 当前节点是否有指定 name 的属性	Boolean	All
hasAttributeNS(namespace, name) 当前节点是否有指定namespace,name的属性	Boolean	All
hasAttributes() 当前节点是否有属性	Boolean	All
removeAttribute(name) 删除当前节点给定name的属性。	-	All
removeAttributeNS(namespace, name)		

删除当前节点给定namespace,name的属性。

removeAttributeNode(name) 删除当前节点给定name的属性节点	–	All
setAttribute(name, value) 设定当前节点给定name的属性值.	–	All
setAttributeNS(namespace, name, value) 设定当前节点给定namespace,name的属性值.	–	All
setAttributeNode(name, attrNode) 设定当前节点给定name的属性值节点。	–	All
setAttributeNodeNS(namespace, name, attrNode) 设定当前节点给定namespace,name的属性值节点。	–	All

上表是 mozilla 文档中 element 的对于 attribute 的相关操作函数。我们可以看出对于属性的操作只有取值，设值，删除，判断四个方面。每个方面都有对属性值，属性节点和带有命名空间的操作。

但是对于大多数使用，我们只会用到getAttribute(name)、setAttribute(name, value)、removeAttribute(name)这三种。而jquery正好提供了这三种的实现。它主要的功能就是解决IE、FF等等的兼容问题。比如opacity属性。还有一些html中标签的属性名是元素的属性的简写，如for—htmlFor。我们现在调用Jquery的方法只要直接用简写的for就可以，jquery会完成它们对应的转换。

Jquery不光在这里给我们提供支援了方便，就是名字上也提供了方便。它把getAttribute(name)、setAttribute(name, value)长且难记的方法名统一在attrr的函数中。

```
// attr(properties)
// 将“名/值”形式的对象设置为所有匹配元素的属性。
// attr(name)
// 取得第一个匹配元素的属性值。通过这个方法可以方便地从第一个匹配元素中获取一个
//属性的值。如果元素没有相应属性，则返回 undefined 。
// attr(key,value)
// 为所有匹配的元素设置一个属性值。$("img").attr("src","test.jpg");
// attr(key,fn)
// 为所有匹配的元素设置一个由这个函数计算的值做属性值。
attr : function(name, value, type) {
    var options = name;// 支持"名/值"形式的对象和string
  if (name.constructor == String)
     if (value === undefined)
        // 调用curCss,attr静态方法返回本对象第一个元素的name的属性值
        return this[0] && jQuery[type || "attr"](this[0], name); ①
     else {// 设定属性值，把name String 转换成"名/值"对象，便于统一的处理
            options = {};
            options[name] = value;
```

```
        }
        return this.each(function(i) {// 为每个元素的指定的name属性设值
             for (name in options)
            // value=options[name],可以是Fn(index),this-->each elem
            jQuery.attr(type ? this.style : this, name, jQuery    ②
                    .prop(this, options[name], type, i, name));
            });
    },
```

上面的代码的①是一个取值操作，因为 type 是内部使用的参数，type 至目前只能是 curCSS，也就是如果传入 type 的值，那就是调用 jQuery.curCSS 在元素的 CSS 中去找到对应的属性，如 hegth,width 等。或者就是调用 jQuery.attr 在元素中找到对应名字的属性。注意 CSS 中的属性和元素的属性是不一样，CSS 是通过元素的属性 class 或 style 来设定的。

在代码的②处也是调用 jQuery.attr 进行设值，不过它会先调用 jQuery.prop 对传入的 value 进行计算，如 value 是函数，就取函数返回值。如果 value 为数字且是为元素的 CSS 设属性值（如 height）的话，说明其没有指定单位，prop 会加上 px 作为默认的单位。

我们先看一下 jQuery.prop 的代码：

```
// 根据指定元素(elem)的指定的name来修正value值，如加px,exec Fn.
  prop : function(elem, value, type, i, name) {
      if (jQuery.isFunction(value))// value=Fn
          value = value.call(elem, i);// 得到Fn的返回value
       // 对于element的style中CSS属性，对需要加上单位的加上px单位
      return value && value.constructor == Number && type == "curCSS"
                && !exclude.test(name) ? value + "px" : value;
      },
```

jQuery.Prop 返回修正后的 value，再回到 attr 中代码的②处看一下 jQuery.attr，它完成取值和设值的功能。

```
  // 为给定的 elem 的 name 属性设定 value 值或取 elem 的 name 属性值
  attr : function(elem, name, value) {
     // 文本，注释节点不处理                                          ①
    if (!elem || elem.nodeType == 3 || elem.nodeType == 8)
          return undefined;
    var notxml = !jQuery.isXMLDoc(elem),//不处理xml文档的
          set = value !== undefined,// 取值还是设值?
          msie = jQuery.browser.msie;//ie
  // 兼容的处理,标签中的属性名是元素属性名的简化：for--htmlFor，
  //标签中的属性名全部是小写，而元素属性名采用lamb形式：rowspan : "rowSpan"
  name = notxml && jQuery.props[name] || name;//可以看出&&、||的用法。

 if (elem.tagName) {//是元素                                        ②
```

```
      var special = /href|src|style/.test(name);// 要特殊处理
        // 对于safari的特殊处理
      if (name == "selected" && jQuery.browser.safari)
                  elem.parentNode.selectedIndex;
      if (name in elem && notxml && !special) {// 通过DOM 0方式进入属性③
           if (set) {// 改变属性但IE中的type类型的元素不能改变
                if (name == "type" && jQuery.nodeName(elem, "input")
                              && elem.parentNode)
                        throw "type property can't be changed";
            elem[name] = value;
            }
           // 对于attr(form,name)，取的是form[name].value
        if (jQuery.nodeName(elem, "form")&& elem.getAttributeNode(name))
                  return elem.getAttributeNode(name).nodeValue;      ④
    return elem[name];
      }

    // 对style进行属性的操作
    if (msie && notxml && name == "style")                              ⑤
         //再次调用本函数，执行第二部分的代码，设定style中cssText.
          return jQuery.attr(elem.style, "cssText", value);
    // IE会报错 see #1070  "" + value转换为字符串
    if (set) elem.setAttribute(name, "" + value); //dom1                ⑥
    var attr = msie && notxml && special//href|src|style 在IE要特殊处理
              ? elem.getAttribute(name, 2): elem.getAttribute(name);
              // 不存在的属性返回null时,改成返回undefined
    return attr === null ? undefined : attr;
}
// 元素的CSS的属性，也就是当elem参数是元素的style时。。。                  ⑦
// IE 使用 filters for opacity
if (msie && name == "opacity") {                                        ⑧
    if (set) {// IE opacity 要层的支持
        elem.zoom = 1;
        // 设 alpha filter 来设定 opacity
     elem.filter = (elem.filter || "").replace(/alpha\([^)]*\)/,"") +
          ((parseInt(value) + '' == "NaN"? "": "alpha(opacity=" + value
          * 100 + ")"));
     }
    //找到opacity的值并返回。
   return elem.filter && elem.filter.indexOf("opacity=") >= 0?
     (parseFloat(elem.filter.match(/opacity=([^)]*)/)[1])/100)+'': "";
  }
// lamb字的支持
name = name.replace(/-([a-z])/ig, function(all, letter) {          ⑨
```

```
            return letter.toUpperCase();});
if (set) elem[name] = value;
return elem[name];
},
```

jQuery.attr函数分成三个部分，第一部分②处之前的代码，它是处理一些准备工作，如标签属性名与元素的属性名之间的对应。第二部分是②~⑦，它主要完成对元素的属性进行设值或取值。③~⑤首先通过Dom 0 的方式（elem[name]）来，如果不能完成的话，就采用⑤~⑦的Dom1 的方式通过getAttribute(name)、setAttribute(name, value)来进行操作。这种操作相对Dom0 的方式，可以对XML进行操作。而且还能对Dom1 的href|src|style属性进行操作。

在⑤处，我们可以看出 IE 中 Style 属性要进行特殊的处理。这个处理就是第三部分的的任务，位处于⑦之后的代码，对于元素的 CSS 属性进行取值或设值。在⑨处，我们可以看到 CSS 属性名只支持 lamb 字的形式。

jQuery.attr 不但可以完成对元素的属性设/取值，还可以能 CSS 进行设/取值。

对于属性的操作，jquery 还提供了 removeAttr。它通过调用 Element ‘s removeAttribute()来完成对 this 中每个元素都删除属性。

```
// 一组对元素attr,class等进行操作的函数
jQuery.each( {
    removeAttr : function(name) {// 除去元素的一个属性
           jQuery.attr(this, name, "");
           if (this.nodeType == 1)
               this.removeAttribute(name);//this==dom element
        },
        .. .. ..
    }, function(name, fn) {
        jQuery.fn[name] = function() {
            return this.each(fn, arguments);
        };
    });
```

5.1.2 Class

在开发过程中，对元素的 class 进行操作是经常的事情，如为元素增加一个 class 或删除一个 class 或对一个 class 进行 toggle 操作。Jquery 提供了三个方法 addClass、removeClass、toggleClass 用来完成对 class 的操作。

```
// 一组对元素attr,class等进行操作的函数
jQuery.each( {
        addClass : function(classNames) {// 为元素增加一些classNames
```

```
            jQuery.className.add(this, classNames);
        },
        removeClass : function(classNames) {// 除去元素的一些classNames
            jQuery.className.remove(this, classNames);
        },
        toggleClass : function(classNames) {// 开关该class，
            jQuery.className[jQuery.className.has(this, classNames)
                    ? "remove" : "add"](this, classNames);
        },
    }, function(name, fn) {
        jQuery.fn[name] = function() {
            return this.each(fn, arguments);
        };
    });
```

上面的代码简单，它们调用jQuery. className中的add或remove方法：

```
    // 一组内部使用的Class操作函数
    className : {
        // 为元素增加classNameS
      add : function(elem, classNames) {// 多个className,空格分开
        jQuery.each((classNames || "").split(/\s+/),
          function(i, className) {
            if (elem.nodeType == 1
                    && !jQuery.className.has(elem.className,className))
            elem.className += (elem.className ? " " : "") + className;
            });
        },
    // 为元素除去classNames
    remove : function(elem, classNames) {
      if (elem.nodeType == 1)// 元素
        elem.className = classNames != undefined ? jQuery.grep(
          elem.className.split(/\s+/), function(className) {// 过滤
                return !jQuery.className.has(classNames,className);
            }).join(" ") : "";
        },
    // 元素有没有className?
    has : function(elem, className) {
      return jQuery.inArray(className, (elem.className || elem)
            .toString().split(/\s+/)) > -1;
        }
},
```

jQuery.className.has 方法先把 elem.className 分成多个 class(如果有多个的话)，再判断参数 className 在数组中的位置来判断元素是否包含指定的 class。jQuery.className.add 先判断元素是不是含有指定的 class，没有话就追加。

jQuery.className.remove 正好相反。

Jquery 还提供了一个 hasClass 用来判断其集合的元素是否含有指定的 class,如果有一个含有的话，就返回 true。

```
/ 检查当前的元素是否含有某个特定的类，如果有，则返回 true
 hasClass : function(selector) {
    return this.is("." + selector);
 },
```

5.1.3 expando (data)

有些时间我们需要保存于一些数据，这些数据与 dom 元素有关。比如我自定义的事件。当然有很多方法来实现。其中有一个环节，我们是不可以少的，就是元素和数据的位置联系起来。采用 dom 元素的 name 或 ID 做为储存的名字行不行，可以，但是不是每个元素都有名字或 Id 的。而且这个 name 或 ID 是经常用到的属性。耦合性有点高。如果能提供一个用户不知道的又不会用的属性来做为两者之间的关系是多好。用户根本不要去关心这个关连的环节。只要知道怎么用就可以。

Jquery 给 dom 元素扩展了 expando 属性。Expando 还是有可能会属性冲突的，jquery 采用了动态生成的常量，它对于每个客户端的浏览器都不一样，但是对运行 jquery 的的整个生命周期就是常量，不变。我们要的就是这样。它定义了一个 var expando = "jQuery" + now()。只要通过 Jquery[expando]就可以找到属于这个生命周期的 expando 常量。

现在 Jquery[expando]就是元素的扩展的属性名。只要在这个属性中分配一个全局唯一的的 Id，同时比存储的地方也指定这个 Id,那么 dom 元素就和数据的储存联系起来。

Jquery 采用 cache : {}做为自定义的数据的储存位置。每一个不同的 dom 元素，要保存数据的话，都在这个里建立相对于的 Id 的 key/value 对。如 cache={Dom1_Id:{},Dom2_id:{}}。如我们在合用$(dom1).data(xx,yyy)。那么其 cache={Dom1_Id:{xx:yyy} }。这样我们可以为元素保储众多的又不冲突的数据。这个在事件应该是很多的。

接下我们看一下 jquery 提供的 Data 的方法

```
// data(name,value) 在元素上存放数据，同时也返回 value。
// 如果jQuery集合指向多个元素，那将在所有元素上设置对应数据。
```

```
    // data(name)返回元素上储存的相应名字的数据，用data(name, value)来设定。
    // 如果jQuery集合指向多个元素，那将只返回第一个元素的对应数据
    data : function(key, value) {
     var parts = key.split(".");
     parts[1] = parts[1] ? "." + parts[1] : "";
    if (value === undefined) {// 取值
            // 加上"!"表明是自定义的事件。在系统扩展的事件用到。
            // 执行getDataxx的定义事件的所有处理函数。返回处理的结果集。
            //如Ext组件的事件机制。这里主要是UI设计之前。
        var data = this.triggerHandler("getData" + parts[1] + "!",
                    [parts[0]]);
        if (data === undefined && this.length)
                data = jQuery.data(this[0], key);
        return data === undefined && parts[1] ? this.data(parts[0]) : data;
        }
   else { //  设值
        return this.trigger("setData" + parts[1] + "!", [parts[0], value])
                    .each(function() {jQuery.data(this, key, value);
                    });
        }
    },
```

上面的代码中的事件先可以不去用研究。这是自定义事件的一种方法。除了这个就是调用了 jQuery.data 来完成设值和取值。

```
//没有data时，就初始化，或取data,有就保存
    data : function(elem, name, data) {
            elem = elem == window ? windowData : elem;
            var id = elem[expando];
            if (!id)//为元素扩展一个全局的id.
                id = elem[expando] = ++uuid;
            if (name && !jQuery.cache[id])
                jQuery.cache[id] = {};//生成cache
          //防止覆盖已经命名的cache采用没有定义的values
            if (data !== undefined)
                jQuery.cache[id][name] = data;
           //cache={id1:{name1:data}}
            //返回命名的cache data
            return name ? jQuery.cache[id][name] : id;

        },
```

jQuery.data 如果只有一个参数 elem。其作用是就取得该元素的 expando 常量的 uuid 属性值。如果之前没有分配，就分配一个唯一的 UUID，之后返回该值。通常一个参数做为初始化元素的扩展 expando 属性而用的。

对于二个参数，elem,name，那就是从 cache:{elem_uuid:{name:vaue}}取 name

对应的 value。如果 cache 没有对应的 uuid，就先初始化，在 cache 分配其对应的空间，之后返回空对象。如果三个参数都有，就是为 cache 中 name 设值。这个 data 可以是任何对象。

内存泄漏一直都是程序的话题，有了增加数据，为了不内存泄漏，在不使用的时候，一定要 removeData：

```
// 在元素上移除存放的数据 与$(...).data(name, value)函数作用相反
removeData : function(key) {
    return this.each(function() {
        jQuery.removeData(this, key);
    });
},
```

看一下: jQuery.removeData:

```
removeData : function(elem, name) {
        elem = elem == window ? windowData : elem;
        var id = elem[expando];
        if (name) {//remove指定名字的
            if (jQuery.cache[id]) {
                //remove name对应的数据
                delete jQuery.cache[id][name];

              //在cache[id])中找不到，说明该元素没有事件
                //那么除去该元素，也就是全局的guid：{}
                name = "";
                for (name in jQuery.cache[id])
                    break;
                if (!name)
                    jQuery.removeData(elem);
            }
        } else {
            try {//remove元素的扩展的expando属性
                delete elem[expando];
            } catch (e) {
                //IE必须要调用removeAttribute在remove
                if (elem.removeAttribute)
                    elem.removeAttribute(expando);
            }
            //在cache中除去uuid：{}
            delete jQuery.cache[id];
        }
    },
```

RemoveData 有二个参数，如果只有一个 elem 的参数，那么就把这个扩展的

属性除去，同时还把 cache 中与扩展 expando 对应的所有的数据都删除。

如果传入两个参数，先删除指定的 name 对应的数据。如果 elem 范扩展的属性 expando 对应的 uuid 是空对象的话，也把这个给删除。如：cache:{elem_uuid:{ }}—> cache:{ }。

5.2 dom 元素的 CSS

对于 web 的应用，控制页面的显示效果是必不可少的一项工作，元素的显示就是通过 CSS 来实现的。CSS 可以控制元素的宽度，高度，位置，可见性等等。

5.2.1 CSS 属性

对于页面样式控制，我们可以事先通过 class 文件或元素的 style 的属性来定好。但是有的时候我们也需要动态去改变样式，比如我想每点击一次，元素的高度就变高一点。这个我们就得动态通过 javascript 来设定。一般的情况我们会采用 elem.style.heigth=elem.style.heigth+1。这只是一个简单的情况，如果我设的 CSS 属性很多，这样就很烦琐。

Jquery 的 css 的函数就提供了能 key/value 的形式来提供多样式属性。

```
// css(name)
// 访问第一个匹配元素的样式属性。
// css(name,value)
// 在所有匹配的元素中，设置一个样式属性的值。数字将自动转化为像素值
// css(properties)
// 把一个“名/值对”对象设置为所有匹配元素的样式属性。这是一种在所有匹配的元素上设置大量样式属性的最佳方式。
css : function(key, value) {
    if ((key == 'width' || key == 'height') && parseFloat(value) < 0)
        value = undefined;// 忽略负数
    return this.attr(key, value, "curCSS");// 调用了curCSS方法
},
```

上面的代码中可以看出其还是调用 attr 来完成任务。在 attr 的 type 参数提供了 curCSS 值。在 5.1.1 节上，我们可以看出如果是取值就是采用 jquery. curCSS, 如果是设值还是采用 jQuery.attr 来完成。Jquery.attr 已经分析。我们看看 jquery. curCSS 怎么从样式中取出值？设值和取值在 CSS 中是不一样，也就是为什么设值能统一在 jQuery.attr，而取值不能。在 CSS 中取值，要分成二种情况，一是从 elem.style 中取值。第二种是已经计算过的 CSS 属性值，这个就包含从 class 得到

样式属性值。因为一般来说 style 中的属性优先级高于 class。所以可以先从 style 中然后再从已经计算的 CSS 属性中找。不过在 FF 中是通过 defaultView.getComputedStyle 来取得属性值，而 IE 是通过 elem.currentStyle 来获得的。如果在 class 中采用了!import。那么 force 就应该为 true。直接从计算过的属性中找才是当前的正在使用的属性。

```
  //找elem的对应name的CSS的属性。
 curCSS : function(elem, name, force) {
    var ret, style = elem.style;
    // elem的属性值被破坏时
    function color(elem) {//仅仅是为了safari
        if (!jQuery.browser.safari) return false;
        // 从defaultView 取得元素的已经计算艻的的样式。
        var ret = defaultView.getComputedStyle(elem, null);
        return !ret || ret.getPropertyValue("color") == "";
       }
    // IE 中opacity 不兼容
    if (name == "opacity" && jQuery.browser.msie) {
        ret = jQuery.attr(style, "opacity");//调用attr,设定IE opacity
        return ret == "" ? "1" : ret;// 1是100%的显示
        }
   // Opera的display bug修正, 见 #2037
    if (jQuery.browser.opera && name == "display") {
        var save = style.outline;
        style.outline = "0 solid black";
        style.outline = save;
    }
      //http://www.w3.org/TR/DOM-Level-2-Style
      ///css.html#CSS-DOMImplementationCSS
     //http://developer.mozilla.org/en/docs/DOM:CSS
     //http://developer.mozilla.org/en/docs/CSS:float
    if (name.match(/float/i))// 标签中float是通过styleFloat取值的
        name = styleFloat;
    if (!force && style && style[name])                                    ①
        ret = style[name];// 取值
        //Returns computed style of the element. Computed style represents
        //the final computed values of all CSS properties for the element.
     else if (defaultView.getComputedStyle)                                ②
    {// 看看defaultView的已经计算过的CSS  defaultView一般==window
        if (name.match(/float/i))   name = "float";
        // 转换成lamb,如addMethod变成add-method
        name = name.replace(/([A-Z])/g, "-$1").toLowerCase();
        //window.getComputedStyle(element, pseudoElt);
        //pseudoElt is a string specifying the pseudo-element to match.
```

```
        //Should be an empty string for regular elements
    //http://developer.mozilla.org/en/docs/DOM:window.getComputedStyle
        var computedStyle = defaultView.getComputedStyle(elem, null);
        if (computedStyle && !color(elem))
        //http://www.w3.org/TR/DOM-Level-2-Style/
        //css.html#CSS-CSSStyleDeclaration
            ret = computedStyle.getPropertyValue(name);                    ③
        else {// Safari没有正确地报道，会提示none elements are involved    ④
            var swap = [], stack = [], a = elem, i = 0;
            // 找到所有的父节点display为none的节点。
            for (;a && color(a); a = a.parentNode) stack.unshift(a);
            for (;i < stack.length; i++)//显示它们
                if (color(stack[i])) {
                    swap[i] = stack[i].style.display;
                    stack[i].style.display = "block";
                }
        //有的浏览器只有在display的情况下才计算CSS的值
         ret = name == "display" && swap[stack.length - 1] != null
                    ? "none": (computedStyle && computedStyle
                        .getPropertyValue(name))|| "";
        //恢复改变display的节点为none
        for (i = 0;i < swap.length; i++)
            if (swap[i] != null) stack[i].style.display = swap[i];
        }
    // We should always get a number back from opacity
    if (name == "opacity" && ret == "")ret = "1";
} else if (elem.currentStyle) {// 元素的currentStyle,在IE中。      ⑤
    //lamb字
    var camelCase = name.replace(/\-(\w)/g, function(all, letter) {
            return letter.toUpperCase();    });
    ret = elem.currentStyle[name] || elem.currentStyle[camelCase];
    // From the awesome hack by Dean Edwards
// http://erik.eae.net/archives/2007/07/27/18.54.15/#comment-102291
// If we're not dealing with a regular pixel number
// but a number that has a weird ending, we need to convert it to pixels
  if (!/^\d+(px)?$/i.test(ret) && /^\d/.test(ret)) {                ⑥
        // Remember the original values
        var left = style.left, rsLeft = elem.runtimeStyle.left;
    // Put in the new values to get a computed value out
    elem.runtimeStyle.left = elem.currentStyle.left;
    style.left = ret || 0;
    ret = style.pixelLeft + "px";
    // Revert the changed values
    style.left = left;
```

```
    elem.runtimeStyle.left = rsLeft;
    }
}
return ret;
},
```

上面代码中①处是通过 style[name]来获得 style 中 CSS 值。如果 style 没有设定这个样式属性或显示指示不从 style 中获取的话，那么从就是计算过的 CSS 中取值。在 FF 系列中会执行②处中间的代码，在 IE 中会执行⑤的代码。

在③处是通过 defaultView.getComputedStyle(elem, null).getPropertyValue(name)来获取属性值，如果没有取到值，很有可能是元素的在文档中没有显示。有的浏览器不会计算它的属性。在④为了取得没有显示出来的元素的属性值而设定的。其先让元素显示出来（如果是父辈元素没有显示，也全部显示），之后计算出属性值取值。再之后恢复其为不显示。

在⑤处是通过 elem.currentStyle[name]来获取属性值。对于为数值的值，通过⑥计算其基于 px 的数值值。

5.2.2 width&heigth

对于元素的宽度和高度，dom 元素提供了 client(clientHeight,clientWidth)、offset(offsetHeight,offsetwidth)、scroll(srollHeight,srollWidth)三种方式，这三种有什么区别呢？client=content+padding。Offset=content+padding+border。Scroll 的宽度和高度都是没有经过 scroll 的原始宽度和高度。也就是这个一般会大于现在显示的尺寸。

Jquery 也提供了三种相关的宽度和高度。Height", "Width"是元素的 content 的宽度和高度。innerHeigh, innerWidth 是在元素的内容之上加上 padding。其实就是 clientHeight,clientWidth。outerHeigh、outerWidth 是在元素的内容上加上 padding、border、margin。

Jquery 的这三类方法比元素的方法的好处在于它们能测量不可见的元素的宽度和高度。

另外 document.body 的值在不同浏览器中有不同解释（实际上大多数环境是由于对 document.body 解释不同造成的，并不是由于对 offset 解释不同造成的）document.documentElement 是兼容的。

```
// 为jQuery对象注册height,width方法
//取得第一个匹配元素当前计算的高（宽）度值（px）。或设值
//在 jQuery 1.2 以后可以用来获取 window 和 document 的高（宽）
```

```
jQuery.each(["Height", "Width"],
  function(i, name) {
    var type = name.toLowerCase();
    jQuery.fn[type] = function(size) {// window的宽度和高度
     return this[0] == window ? (// window的宽度和高度   ①
             jQuery.browser.opera&& document.body["client" + name]
          || jQuery.browser.safari&& window["inner" + name]
          || document.compatMode == "CSS1Compat"&&
             document.documentElement["client"+ name]
          || document.body["client"+ name])
        : this[0] == document ? (// document的宽度和高度   ②
              Math.max(Math.max(
                          document.body["scroll"+ name],
                      document.documentElement["scroll"+ name]),
                  Math.max(
                       document.body["offset"+ name],
                        document.documentElement["offset"+ name])))
        : (size == undefined ? (// 第一个元素的的宽度和高度     ③
             this.length ? jQuery.css(this[0], type) : null)
                 : // 设定当前对象所有元素宽度和高度
             this.css(type, size.constructor == String?
                  size: size+ "px"));
      };
    });
```

在上面的代码中可以看出"Height", "Width"分成三个部分，①处是对 window 的宽高取值,这其实就是 document 的 client 的宽高度。document.documentElement 指的是<html>，而 document.body 指的是<body>它们两者之间的区别不大。CSS1Compat 的模式下，body 外的元素都不会计算宽高的。

②是对 document 求宽高，它的的值可能会大于 window。因为 offset 比 client 多了一个 border 的尺寸。而且 document 还会取比 offset 大的 scroll。

③是取或设其它元素的宽高。取值是通过 jQuery.css 来完成的，而设值是通过 this.css 来完成的，这个在 5.2.1 中讲过。接下看看 jQuery.css。

```
// 取得elem的name的属性值
 css : function(elem, name, force) {
    // 对元素的宽度高度修正
    if (name == "width" || name == "height") {
      var val,props = {position : "absolute",
                       visibility : "hidden",display : "block"},
          which = (name == "width" ? ["Left", "Right"] : ["Top","Bottom"]);
   function getWH() {// 求元素的实际高度，宽度
       offsetWidth=padding+border+element
        val = name == "width"? elem.offsetWidth: elem.offsetHeight;
```

```
        var padding = 0, border = 0;
        jQuery.each(which, function() {
            padding += parseFloat(   // paddinLeft,paddingRight
                    jQuery.curCSS(elem, "padding" + this, true))|| 0;
            border += parseFloat(// borderLeftWidth,borderRightWith
                 jQuery.curCSS(elem, "border"+ this + "Width", true))|| 0;
                       });

    //http://msdn.microsoft.com/en-us/library/ms530302(VS.85).aspx
    //http://msdn.microsoft.com/en-us/library/ms534304(VS.85).aspx
//offsetwidth-paddinLeft-paddingRight-orderLeftWidth-borderRightWith
    val -= Math.round(padding + border);//height也同样要减去。
    }
//可见就取得实际元素的w,h。除padding,border
  if (jQuery(elem).is(":visible"))getWH();
// 元素看不到的情况下，通过使元素暂时为绝对定位，display:block等来取高度或宽度
  else jQuery.swap(elem, props, getWH);
return Math.max(0, val);
    }
return jQuery.curCSS(elem, name, force);

   },
```

jQuery.css 大部分是调用 jQuery.curCSS 来完成工作的，它只是完成了对元素的宽高进行修改。因为 CSS 中的宽高度的属性在 IE、FF 的浏览器是完全不一样的。IE 中是 content、padding、border 的尺寸的总和，而 FF 中仅仅是 content 的尺寸。

在 jQuery.css 为了保证兼容性，把标准的 content 的尺寸作为每个浏览器的尺寸。每种浏览器的 CSS Box 模式是不一样，但是它们的元素的 offsetHeight（width）是一样的。取得元素的宽高只要在 offset 上面减去 padding 和 border 就可以了。

对于不可见的元素，浏览器不计算其尺寸，可以通过先可见计算其 size 然后恢复。这个的实现在 jQuery.swap(elem, props, getWH)中。

```
// 改变elem的options指定的属性以便执行callback中使用，完成之后又恢复原定属性。
 swap : function(elem, options, callback) {
            var old = {};
            for (var name in options) {// 替换elem.style中的属性
                old[name] = elem.style[name];
                elem.style[name] = options[name];
            }
            callback.call(elem);
            for (var name in options) // 重新换回原来的属性
        elem.style[name] = old[name];
```

```
        },
```

接下来我们分析 jquery 的。innerHeigh, innerWidth、outerHeigh、outerWidth。它们是在一起实现的。

```
// Create innerHeight, innerWidth, outerHeight and outerWidth methods
jQuery.each(["Height", "Width"], function(i, name) {
    var tl = i ? "Left" : "Top", // top or left i=0时，为false
        r = i ? "Right" : "Bottom"; // bottom or right
 jQuery.fn["inner" + name] = function() {// inner的尺寸          ①
          return this[name.toLowerCase()]() + num(this, "padding" + tl)
                 + num(this, "padding" + br);
        };
 jQuery.fn["outer" + name] = function(margin) {//outer的尺寸      ②
        eturn this["inner" + name]()
                 + num(this, "border" + tl + "Width")
                 + num(this, "border" + br + "Width")
                 + (margin ? num(this, "margin" + tl)
                        + num(this, "margin" + br) : 0);
        };
    });
```

在①处是求元素的 innerHeight, innerWidth。在②求 outerHeight 和 outerWidth。Inner 等于元素的宽高度加各自的 paddng。而 out 还要加上 border 和 margin。这里调用 num（）来得到元素的 CSS 的属性值的大小（只能是数值型的）

```
// Helper function used by the dimensions and offset modules
function num(elem, prop) {
return elem[0] && parseInt(jQuery.curCSS(elem[0], prop, true), 10) || 0;
 }
```

5.2.3 position

在给元素定位之前，我们首先要了解一些 CSS 定位相关的知识。

在 CSS 中关于定位的内容是：position:relative | absolute | static | fixed 。static 没有特别的设定，遵循基本的定位规定，不能通过 z-index 进行层次分级。relative 不脱离文档流，参考自身静态位置通过 top,bottom,left,right 定位，并且可以通过 z-index 进行层次分级。absolute 脱离文档流，通过 top,bottom,left,right 定位。选取其最近的父级定位元素，当父级 position 为 static 时，absolute 元素将以 body 坐标原点进行定位，可以通过 z-index 进行层次分级。

fixed 固定定位，这里他所固定的对像是可视窗口而并非是 body 或是父级元素。

可通过 z-index 进行层次分级。CSS 中定位的层叠分级：z-index: auto | namber

relative | absolute | static | fixed 这四种定位的方式不一样，我们要找到元素的位置的方法也会随之不一样。

Dom 元素提供了三种方式来定位元素：offset，scroll，Client，

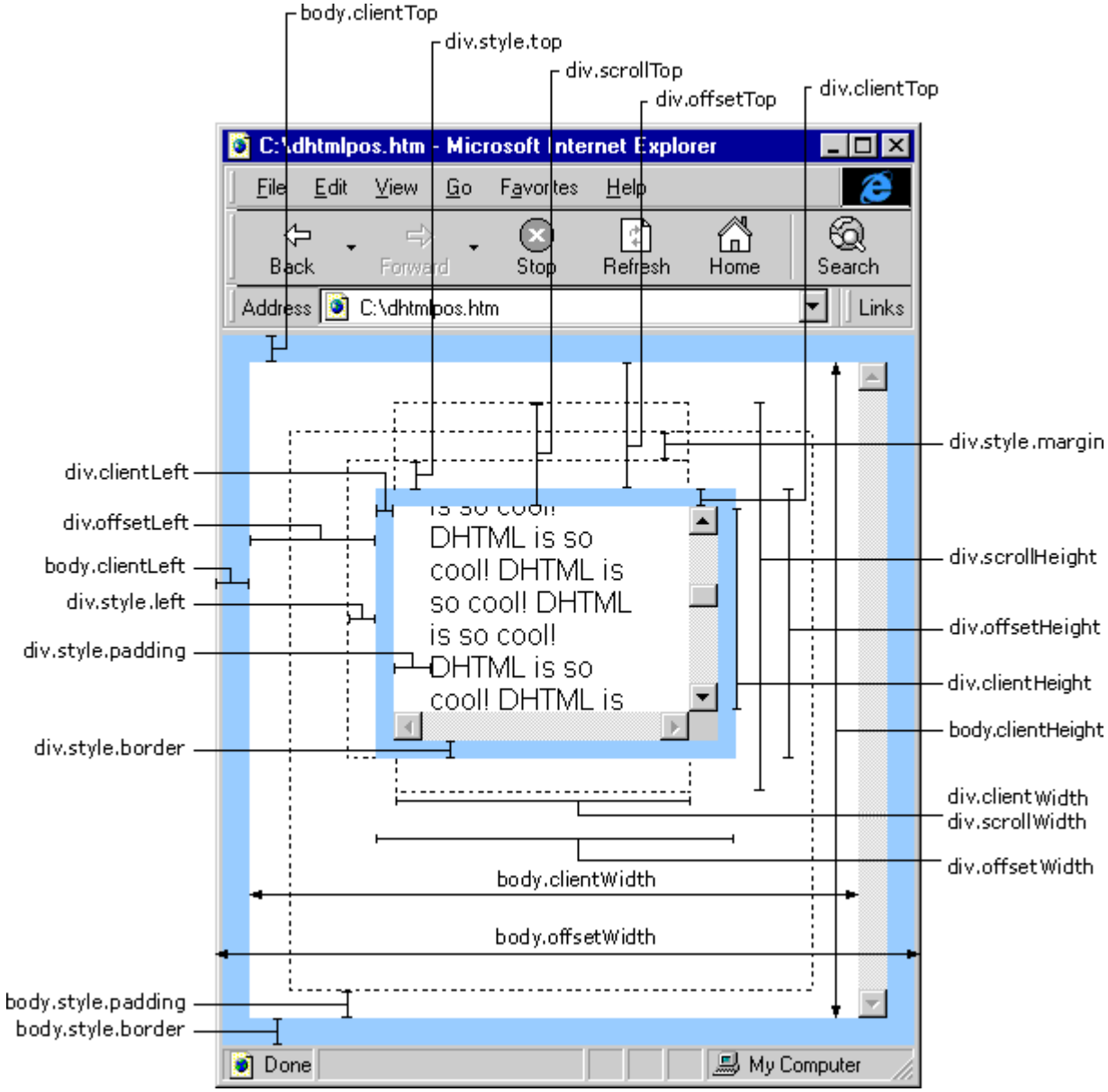

图 转自（http://www.cnblogs.com/believe3301/archive/2008/07/19/1246806.html）

Dom 元素对于 offset 提供了 offsetParent、offsetTop、offsetLeft、offsetWidth、offsetHeight 五个方法来定位于元素的相对位置。

offsetParent 是指当前元素的相对定位的元素。在 IE 和 FF 中定义和解释不一样。在 IE 中定义为获取定义对象 offsetTop 和 offsetLeft 属性的容器对象的引用。大多数时候 offsetParent 返回 body 元素。在 IE5 中，td 的 offsetParent 是 table。可以看出 IE 中的相对定位与绝对定位的区别不大。都是相对于最上层的元素来定位。在 FF 中获取文档层次中最近的元素。如果这个元素没有定位，那么就根

元素。

offsetParent、parentNode（IE:parentElement）都是指元素的父节点。它们的针对的目标是不一样，功能也不一样。parentNode 就是取文档层次中包含该节点的最近节点(直接的父节点)。在FF中对于Attr, Document, DocumentFragment, Entity, 和 Notation 这些父节点，其 parentNode 返回 null。还有如果没有附加到文档树的元素也是返回 null。

offsetParent 是指可视的父节点。如 <body><form><input type=’’’text’ id=’AA’/></form></body>。AA 的 offsetParent 是 body，而 parentNode 则是 form。在 IE 中一般都是 body。

offsetLeft 和 offsetTop 是指当前元素 left 或 top 边到 offsetParent 的 left 或 top 边的距离，包含了当前元素的 margin 和其 offsetParent 的 padding。不包含 offsetParent 的 border 的宽度。

offsetWidth、offsetHeight 与 offsetLeft、offsetTop 的相对 offsetParent 的方式不一样，它们就是当前元素自身的宽度或高度。它包含 border、padding、scrollBar(显示的话)和内容的 size(CSS 中设定的元素的高度，IE 中 CSS size 指的是包含 border 的内容大小)。

分析了 offset，我们可以发现 offsetLeft、offsetTop 与 CSS 中 top,left 的属性有相通性，offsetLeft、offsetTop 只能取值。而我们可以通过 CSS 中 top,left 的属性来设定一个元素的相对其它元素的位置（绝对定位，就是相对于 body）。

Dom 元素对于 scroll 提供了 scrollWidth、scrollHeigth、scrollTop、scrollLeft。这一组是对于 scroll 的元素进行操作的。Scroll 的 Width、Heigth 是指元素真实的宽度和高度，它包含被 scroll 起来的部分。而 scrollTop、scrollLeft 则是被卷起来的部分的大小。

Dom 元素对于 scroll 提供了 clientWidth、clientHeigth、clientTop、clientLeft。这一组是对于 client 进行操作的。clientWidth、clientHeigth 是元素的内容可视区域的高度或宽度。包含 padding，不包含 scrollbar 、border、margin。可以看出是元素可视的区域。IE，FF 是一样的。clientTop、clientLeft 可以看做是 topborder 或 left border 的大小。

offsetParent的名字的元素能计算相对位移的父节点，那么对于CSS的定位方式，哪一些是可以计算位移呢，能计算元素和其父节点之间的位移量，首先要其父节点能定位。这个定位就是在CSS中能采用top,left来定其在文档的位置。Body是肯定可以的（0，0）。Body也是元素的终结offsetParent（没有找到就是它

了）,absolute、 relative、 fixed都采用可以top,left来定其在文档的位置。也是能计算其位置。而static是不需要top,left来设定其位置， Offset是相对已经定位的元素的位移。元素的offsetParent是其父辈节点中的postiont!= Static的节点。在IE中 http://msdn.microsoft.com/zh-cn/library/system.windows.forms.htmlelement.offsetparent(VS.80).aspx ， 可 以 看 到 其 不 支 持 fixed 的 offsetParent 。 在 mozilla 中 http://developer.mozilla.org/en/DOM/element.offsetParent，可以看出其给出的如果元素没有定位（non-positioned）就是body。

Jquery 针对于 offsetParent 提供了一个改进的方法。它还是在浏览器的 offsetParent 基础之上多加了一个判断的处理，筛选其有可能会是 static 的节点。觉得这样做的意义不大。除了 table,tr,td 之外，浏览器的 offsetParent 是 body 的可能性很大。这是一个不确定的。在使用中是要注意的。

```
//找到this[0]中元素第一个能根据CSS中的top,left能设定位置的父辈节点。
    offsetParent: function() {
        var offsetParent = this[0].offsetParent || document.body;
    while ( offsetParent && (!/^body|html$/i.test(offsetParent.tagName)
            && jQuery.css(offsetParent, 'position') == 'static') )
        offsetParent = offsetParent.offsetParent;
        return jQuery(offsetParent);
    }
```

其实觉得最好的方法还是直接相对于 body 的来定位。这样的定位是确定的。但是浏览器在计算这个值会有点问题，而且每种浏览器的实现方式不一样，很难兼容。Jquery 提供了一个相对于文档的起始位置的 offset 方法。

```
//元素相对于文档的起始位置的offset
jQuery.fn.offset = function() {
    var left = 0, top = 0, elem = this[0], results;
    if ( elem ) with ( jQuery.browser ) {
        var parent    = elem.parentNode,    offsetChild  = elem,
          offsetParent = elem.offsetParent,
         doc      = elem.ownerDocument,
     safari2   = safari && parseInt(version) < 522
                && !/adobeair/i.test(userAgent),
       css        = jQuery.curCSS,
       fixed       = css(elem, "position") == "fixed";

 //在IE中有的元素可以通过getBoundingClientRect来获得元素相对于client的rect.
if ( !(mozilla && elem == document.body) && elem.getBoundingClientRect )
    {//IE  http://msdn.microsoft.com/en-us/library/ms536433.aspx    ①
    var box = elem.getBoundingClientRect();
```

```
// 加上document的scroll的部分尺寸到left,top中。
add(box.left + Math.max(
      doc.documentElement.scrollLeft, doc.body.scrollLeft),
    box.top  + Math.max(
     doc.documentElement.scrollTop,  doc.body.scrollTop));
      //IE中会自动加上2px的border,这里是去掉document的边框大小。

//http://msdn.microsoft.com/en-us/library/ms533564(VS.85).aspx
/The difference between the offsetLeft and clientLeft properties
// is the border of the object
add( -doc.documentElement.clientLeft,
                       -doc.documentElement.clientTop );
} else {                                                    ②
//通过遍历当前元素offsetParents来计算其在文档中的位置（相对于文档的起始位置）
add( elem.offsetLeft, elem.offsetTop );//初始化元素left,top    ③
 //很多浏览器的offsetParent是直接指向body。不过有的是指向最近的可视的父节点。
while ( offsetParent ) { //加上父节点的偏移
    add( offsetParent.offsetLeft, offsetParent.offsetTop );  ④
 // Mozilla系列offsetLet或offsetTop不包含offsetParent的边框。要加上
 //但在在table中又会自动加上。
  if ( mozilla && !/^t(able|d|h)$/i.test(offsetParent.tagName)
          || safari && !safari2 )
    border( offsetParent );//增加offsetParent的border          ⑤
  // 对于CSS设定为fixed相对于client的定位，加上document.scroll.
  if ( !fixed && css(offsetParent, "position") == "fixed" )
             fixed = true;
  //改变子节点变量offsetChild，再改变offsetParent变量的指向。
   offsetChild  = /^body$/i.test(offsetParent.tagName) ?
               offsetChild : offsetParent;
        offsetParent = offsetParent.offsetParent;
    }
// 减去处理每一层不显示的scroll的部分。
// 因为一个元素的size(CSS中指定的）是scroll之前的。
// 如果scroll，offsetLet或offsetTop会包含这部分被卷起的。
while ( parent && parent.tagName
       && !/^body|html$/i.test(parent.tagName) ) {       ⑥
// 如果parent的display的属性不为inline|table，减去它的scroll.
if ( !/^inline|table.*$/i.test(css(parent, "display")) )
    // 减去 parent scroll offsets
    add( -parent.scrollLeft, -parent.scrollTop );
    // 如果overflow != "visible.在Mozilla 中就不会加上border.s
    if ( mozilla && css(parent, "overflow") != "visible" )
           border( parent );
        parent = parent.parentNode;
```

```
            }

    //Safari <= 2，在CSS中position为fiexed或者body的position==absolute，
    //会重复加上body offset。Mozilla在Position! =absolute的时候也会重复
 if ( (safari2 && (fixed || css(offsetChild, "position") == "absolute"))
        ||  (mozilla && css(offsetChild, "position") != "absolute") )
    add( -doc.body.offsetLeft, -doc.body.offsetTop );            ⑦

    //fixed 加上document scroll。
     //因为fixed是scroll的时候也是相对于client不变。所以要加上
    if ( fixed )                                                 ⑧
    add(Math.max(doc.documentElement.scrollLeft, doc.body.scrollLeft),
        Math.max(doc.documentElement.scrollTop,  doc.body.scrollTop));
    }
    results = { top: top, left: left };
}
function border(elem) {
        add( jQuery.curCSS(elem, "borderLeftWidth", true),
jQuery.curCSS(elem, "borderTopWidth", true) );
    }
function add(l, t) {
        left += parseInt(l, 10) || 0;
        top += parseInt(t, 10) || 0;
    }
  return results;
};
```

上面的代码①采用 IE 内部提供了特有的方法来找到相对 body 的 Offset。这样做肯定是提高在 IE 中效率。②③④⑤⑥⑦⑧处是采用通用的处理方法来计算。对于一个元素的 offset,加上其所有 offsetParent 的 offset 和 border。这样就能计算出相对于 body 的 offset。但是这样在 scroll 的情况下是行不通的。因为有 scroll 的卷起来的部分也被计算了进去，对于每个元素都要减去这一部分的大小。

⑧处我们可以看出如果有元素是 fixed 的 position。说明其会随着 documentElement.scroll 而改变位置。因此加上 documentElement.scroll。得出其正确的位置。

Jquery 中的 position 方法是计算当前元素相对于其 offsetParent 的 offset 值。与 dom 元素的 offset 不一样的地方，它是建立在 jquery.offset 的基础之上，同时还不包括其自身的 margin。对于 box 的模式来讲，是 margin 是元素的最外边，而不是 border。

```
jQuery.fn.extend({position: function() {
    var left = 0, top = 0, results;
```

```
    if ( this[0] ) {
        var offsetParent = this.offsetParent(),// Get *real* offsetParent
          offset   = this.offset(),// Get correct offsets
        parentOffset = /^body|html$/i.test(offsetParent[0].tagName)
                    ? { top: 0, left: 0 } : offsetParent.offset();
            offset.top  -= num( this, 'marginTop' );
            offset.left -= num( this, 'marginLeft' );
            parentOffset.top  += num( offsetParent, 'borderTopWidth' );
            parentOffset.left += num( offsetParent, 'borderLeftWidth' );
            results = {
                top:  offset.top  - parentOffset.top,
                left: offset.left - parentOffset.left
            };
        }
        return results;
    },
```

Jquery 还提供了两个关于 scroll 的方法， scrollLeft and scrollTop：

```
// Create scrollLeft and scrollTop methods
jQuery.each( ['Left', 'Top'], function(i, name) {
    var method = 'scroll' + name;
    jQuery.fn[ method ] = function(val) {
        if (!this[0]) return;
        return val != undefined ?   // Set the scroll offset
            this.each(function() {
                this == window || this == document ?
                    window.scrollTo(
                        !i ? val : jQuery(window).scrollLeft(),
                         i ? val : jQuery(window).scrollTop()
                    ) :
                    this[ method ] = val;
            }) : // Return the scroll offset

            this[0] == window || this[0] == document ?
                self[ i ? 'pageYOffset' : 'pageXOffset' ] ||
                    jQuery.boxModel && document.documentElement[ method ]
                    ||document.body[ method ] :
                this[0][ method ];
    };
```

5.3 dom 元素的操作

Dom 的元素提供了一些改变 Dom 文档的层次结构的方法。改变 dom 文档可以归纳为三种，往文档中插入元素（insert），把元素从文档删除(remove)或是把文档的某些元素用其他的元素来代替(update)。

Dom 元素对于插入提供了 insertBefore()和 appendChild()方法。在 IE 中还为 TextNode 提供了 appendData()。提供了三个 insertAdjacentText、insertAdjacentHTML、insertAdjacentElement()方法实现了在元素的前面，后面，内部开始，内部结束的位置上插入。对于删除，提供了 removeChild()方法，Ie 还提供 removeNode(true)实现删除本元素及所有的子元素。据说这个函数会出现内存泄漏。对于 update，Dom 元素提供了 replaceChild()。IE 还提供了 replaceNode()，replaceAdjacentText()方法。

Jquery 也提供了通用的类似的方法。

5.3.1 create

在使用 insert,update 之前都必须要有元素，这个新增加的元素从那里来呢，就是得创建，document 提供了 createElement 来创建一个元素。别外一种方式就是 clone。对原来的元素进行 clone。Dom 元素提供了 cloneNode 的方法。

Jquery 也提供了类似的方法，jQuery.clean 就是完成元素从 HTMLstring 的到 Dom 对象的创建。这个方法比 document. createElement()提供了更好的兼容和更强大的功能。在 3.1 中已经介绍。

对于 cloneNode，jquery 的提供了类似方法如下：

```
// clone当前对象，events表明是否clone事件
clone : function(events) {
   // 对每个元素都进行加工（copy)后的集合重新构建jQuery对象
   var ret = this.map(function() {
      if (jQuery.browser.msie && !jQuery.isXMLDoc(this)) {
      // IE 中cloneNode不能copy 通过attachEvent增加的事件
      // 而innserHTML不能copy一些修改过的属性（仅作为属性储存）如input 的name
      var clone = this.cloneNode(true),
          container = document.createElement("div");
          container.appendChild(clone);
          return jQuery.clean([container.innerHTML])[0];
          }
       else   return this.cloneNode(true);
      });
```

```
    //需要把元素的扩展属性expando设为null.removeData在这里不起作用。
        var clone = ret.find("*").andSelf().each(function() {
            if (this[expando] != undefined)
                this[expando] = null;
        });

    if (events === true)// clone所有事件
        this.find("*").andSelf().each(function(i) {
            if (this.nodeType == 3)  return;
            var events = jQuery.data(this, "events");

            for (var type in events)//为新元素增加clone的事件
                for (var handler in events[type])
                  jQuery.event.add(clone[i], type,
                    events[type][handler],events[type][handler].data);
            });

        return ret;
    },
```

Clone 提供了对于 jquery 对象的 clone，在上面的代码中可以看出不光是 clone 了元素的本身，还为 clone 的元素 clone 了事件。

5.3.2 insert

在 IE 中提供了 insertAdjacentElement()方法，这个方法比 insertBefore()和 appendChild()好用。为了保证兼容性，像 prototype,mootools，Ext 都提供了一个类似的方法，之后在扩展出四个方法。分别在元素的前面或后面，元素的内部开始或内部结束处四个地方插入元素。

Jquery 也一样，提供了 domManip（），这个函数和 insertAdjacentElement 类似，不同的是它采用回调函数做 swhere 的类似参数，可见它更为灵活。

```
  // Dom manipulate操作的函数，对于每个jQuery对象中元素都运行
    // 由callback操作args转化成的Dom 元素集合的函数。
domManip : function(args, table, reverse, callback) {
    var clone = this.length > 1, elems;                    ①
    // 对当前的jquery对象中每个元素都进行操作
    return this.each(function() {
        if (!elems) {// 把args 转化为dom元素数组，追加的内容    ②
            elems = jQuery.clean(args, this.ownerDocument);
                if (reverse)  elems.reverse();// 倒序
                }
```

```
        var obj = this;
    // Ie Table不兼容，要进行特殊处理
 if (table && jQuery.nodeName(this, "table")// 当前元素是table?  ③
                 && jQuery.nodeName(elems[0], "tr"))// 要追加是tr?
   obj = this.getElementsByTagName("tbody")[0]// 没有tbody,创建追加
    || this.appendChild(this.ownerDocument .createElement("tbody"));
 var scripts = jQuery([]);
jQuery.each(elems, function() {//对于参数转化的每一个dom对象
          // 长度大于1，就采用clone。取第一个元素，否则就是本元素
          var elem = clone ? jQuery(this).clone(true)[0] : this; ④

        // 执行所有 scripts 在所有的元素注入之后
    if (jQuery.nodeName(elem, "script"))scripts = scripts.add(elem); ⑤
    else { // 除去内部 scripts，同时保存起来， 为了之后的计算
      if (elem.nodeType == 1)
           scripts = scripts.add(jQuery("script", elem).remove());⑥
            callback.call(obj, elem);                              ⑦
        }
     });
    scripts.each(evalScript);                                      ⑧
});
}

};
```

domManip 的功能不好形容。从名字的意思来看是指是对 dom 元素的操作。其实我们可以理解为它对 args 参数中的每个元素都传到 callback（elem）的回调函数的参数中去。有点像 jquery.each(elems,function(i,elem){})的函数。它的所有的实质的功能都落到回调函数上。与 each 不一样的是，callback(elem)的 this 指向每一个 jquery 对象中的元素。也就是有二个嵌套的 each。一个 each 是对于 jquery 对象的操作，遍历每一个元素进行回调操作。第二个 each 在回调函数中，就是对于每个元素再次遍历 args 参数的元素。进行回调操作。

对于 args 参数，可以是 string，dom 元素，(类)数组等，它采用②处的代码把 args 转换成 dom 元素组成的数组（对于有的 string 转换如 td，它会自动加上 <table><tr>）。②处的转换过程只执行一次。Table 参数是指在对表操作时，会不会把追加<tbody>，这个在③处理。①④的代码是为了防止多次同时运行 args 中的元素引起的冲突。

⑤⑥⑧可以看出，args 的参数中的元素支持 script 和元素内部的 script 运行。最后统一运行这些 script。

Jquery 在这些函数之上提供 append prepend before after 4 种操作。

$(A).append(B)实现把 B 元素插入到 A 元素内部的尾部。, $(A).prepend(B)实现把 B 元素插入到 A 元素内部的开始位。 $(A).before(B)、实现把 B 元素插入到 A 元素前面。$(A).after(B) 现把 B 元素插入到 A 元素后面。appendTo prependTo insertBefore ,insertAfter 正好相反，实现把 A 插入到 B 元素的尾部、前部、开头，后面。

```
// ************************************************************************
    // 该组方法主要是完成把元素插到什么地方，与Ext的DomHelp的功能相似。
    // 在一个元素之前，之后，元素的开始，结束位置

    // 向每个匹配的元素内部追加内容。
    // 这个操作与对指定的元素执行appendChild方法，将它们添加到文档中的情况类似
    //返回还是当前的jquery对象。
    append : function() {
        //对于当前对象的每一个元素都append arguments中转化的第一个dom对象。
        return this.domManip(arguments, true, false, function(elem) {
            if (this.nodeType == 1)

    //http://developer.mozilla.org/en/docs/DOM:element.appendChild
                this.appendChild(elem);
        });
    },
    // 向每个匹配的元素内部前置内容。
    // 这是向所有匹配元素内部的开始处插入内容的最佳方式。
    prepend : function() {// elem =arguments的转化集合中的dom元素
        return this.domManip(arguments, true, true, function(elem) {
    if (this.nodeType == 1)// this=jQuery对象的每个元素（对于tr之类会修正)
                    this.insertBefore(elem, this.firstChild);
            });
    },

    // 在每个匹配的元素之前插入内容。
    before : function() {
        return this.domManip(arguments, false, false, function(elem) {
            this.parentNode.insertBefore(elem, this);
         // this=jQuery对象的每个元素  });
    },

    // 在每个匹配的元素之后插入内容
    after : function() {
        return this.domManip(arguments, false, true, function(elem) {
            this.parentNode.insertBefore(elem, this.nextSibling);
        });
    },
```

```
  // ************************************************************
// 为jQuery对象生成appendTo~replaceAll五个代理函数
// 其功能是把当前的jquery对象的每个元素都插入到每个传入的参数(元素）的一个位置：
// 之前，之后，开始，结束,overwrite
//把所有匹配的元素追加到另一个、指定的元素元素集合中。
//实际上，使用这个方法是颠倒了常规的$(A).append(B)的操作，即不是把B追加到A中，而
//是把A追加到B中。
jQuery.each( {
    appendTo : "append",
    prependTo : "prepend",
    insertBefore : "before",
    insertAfter : "after",
    replaceAll : "replaceWith"
}, function(name, original) {
    jQuery.fn[name] = function() {
        var args = arguments;// 每个参数和每个元素是对应起来的
        // 对当前jQuery中每个元素都进行的操作
        return this.each(function() {
            for (var i = 0, length = args.length;i < length; i++)
                jQuery(args[i])[original](this);// 调用original代理工作
this是元素
            });
    };
});
```

5.3.3 update

Dom 元素提供了 replaceChild()。IE 还提供了 replaceNode()，replaceAdjacentText()方法。而 jquery 也为其提供了类似的方法：

```
    // 将所有匹配的元素替换成指定的HTML或DOM元素。
    replaceWith : function(value) {
        return this.after(value).remove();// this.after(value),this没有
变
    },
```

replaceWidth 是在本对象的所有元素后面加上 value 元素。然后再把当前对象中的所有元素都去掉。相当于用新的元素代替了原来的元素。Jquery 还提供了 replaceAll 实现把参数中的所有元素都用当前对象的元素来 replace，如果当前对象是多个元素，那最终结果当然是最后一个元素。应该为一个元素。replaceAll(elem,[elem].. .. ..)是它的形式。

```
//用匹配的元素替换掉所有 selector匹配到的元素。
```

```
jQuery.each( {replaceAll : "replaceWith"},
  function(name, original) {
    jQuery.fn[name] = function() {
      var args = arguments;// 每个参数和每个元素是对应起来的
      // 对当前jQuery中每个元素都进行的操作
      return this.each(function() {
          for (var i = 0, length = args.length;i < length; i++)
        // 调用original代理工作 this是元素
          jQuery(args[i])[original](this);          });
   };
});
```

5.3.4 remove

对于删除，提供了 removeChild()方法，Ie 还提供 removeNode(true)实现删除本元素及所有的子元素。据说这个函数会出现内存泄漏。Jquery 也提供了两个方法来 remove 元素及释放内存。

```
jQuery.each( {.. .. ..
      remove : function(selector) {// 根据selector除去元素，防内存泄露①
      if (!selector || jQuery.filter(selector, [this]).r.length) {②
            // Prevent memory leaks  this是指dom元素
            jQuery("*", this).add([this]).each(function() {      ③
                jQuery.event.remove(this);                    ④
                jQuery.removeData(this);
            });
            if (this.parentNode)
                this.parentNode.removeChild(this);            ⑤
        }
      },
      empty : function() {// 清除元素的所有子节点
        // 删除当前对象的每个元素的的所有子节点，防止内存泄漏
        jQuery(">*", this).remove();     ⑥
        while (this.firstChild)
            // 删除余留的子节点
            this.removeChild(this.firstChild);   ⑦
      }
   }, function(name, fn) {
      jQuery.fn[name] = function() {
        return this.each(fn, arguments);  ⑧
      };
   });
```

当我们调用 remove 的方法时，先执行⑧处对每一个元素都进行①的回调函数的调用 。selector 就是筛选当前元素是否可以要进行 remove，一般都要的。之后就是找到元素本身及所有的后代元素组合集合。之后对集合中的每一个元素都去掉事件和捆绑的事件。最后通过 parentNode.removeChild 除去节点。

Empty 与 remove 不同的地方在于 empty 只是除去所有的后代节点（不包含本身），而 remove 包含自身。⑥处采用 remove 方法就是除去所有的子节点。⑦处是确保所有的子节点（后代节点）都给去掉了。

5.3.5 wrap

Jquery 的 wrap 的系列的函数可以说是 dom 元素操作中较为复杂的方法。它提供,wrapAll,wrapInner,wrap 三种方法。

wrapAll 是把 jquery 对象中所有元素都包裹在给定的元素最内部元素中(firstChild)。这个包裹之后的元素在 jquery 对象第一个元素的位置之前。同时会把所有其它位置的元素都去掉。也就是说 wrapAll 最好是不要多个元素。因为多个元素就不太像 wrapAll。

```
    // ************************************************************************
    // 一组用于元素标签包裹操作的函数

    // 将所有匹配的元素用单个元素包裹起来
    // 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。
    // 这于 '.wrap()' 是不同的，'.wrap()'为每一个匹配的元素都包裹一次。
    wrapAll : function(html) {
        if (this[0])
    /*
     * html的内容：<p>Hello</p> xxxxxx <p>cruel</p> xxxxxxx <p>World</p>。
     * $("p").wrapAll("<div><b></b></div>");它调用wrapAll（）。
     * 当前jQuery对象(称为A，$("p"))的集合有三个元素：<p>Hello</p>、
     * <p>cruel</p>、<p>World</p>
     * 开始：jQuery(html, this[0].ownerDocument)把html生成Dom对象
     * (通过调用clean)。
     * 第一步:复制生成一个jQuery对象(称为B)。B的集合中有一个元素，其为：
     * <div><b></b></div>的元素。
     * 第二步：把B所有的元素都插在A[0]元素Dom结构之前，A[0]元素没有变，
     * 新的DOm:<div><b></b></div><p>Hello</p>
     * 第三步：找到B对象中所有元素的最内面的节点，如<div><b></b></div>。
     * <b></b>是B中元素最inner Node;
     * 第四步：向所有innerNode内部插入A对象的所有元素,得到
     * <div><b><p>Hello</p><p>cruel</p><p>World</p></b></div>
```

```
     *
     * 如果是html是selector的话，那就只能是一个元素了。如果是传入的参数是数组，
     * jquery对象，那么可能会有多个元素了。
     * 比如 $("p").wrapAll($("td"));那就是在所有td元素的最内部节点插入：
     * <p>Hello</p><p>cruel</p><p>World</p>
     * If child is a reference to an existing node in the document,
     * appendChild moves it from its current position to the new position 。
     *
     * 对元素进行wrap，最好还是只有this[0].因为如果有多个不同的引用，
     * 会导致其它地方的元素的move。
           */
    jQuery(html, this[0].ownerDocument).clone().insertBefore(this[0])
        .map(function() {// 找到当前元素的最下层的子节点，map新构成jquery对象。
                    var elem = this;
                    while (elem.firstChild)
                        elem = elem.firstChild;
                    return elem;
                }).append(this);// this指是调用wrapAll的jQuery对象。
        return this;
    },

    // 将每一个匹配的元素的子内容(包括文本节点)用一个HTML结构包裹起来
    wrapInner : function(html) {

        /*
         * <p>Hello</p><p>cruel</p><p>World</p>
         * $("p").wrapInner("<b></b>");
         * <p><b>Hello</b></p><p><b>cruel</b></p><p><b>World</b></p>
         */
        return this.each(function() {
              // 这里包裹的对象是每个元素的对象的contents()
                    //每一个元素的所有的子节点，且构建于jquery对象。
                    jQuery(this).contents().wrapAll(html);
                });
    },

    // 对于当前的jquery对象的每个元素都执行wrapAll(html)
    wrap : function(html) {
        //与inner不同的是当前的节点。
        return this.each(function() {// 这里包裹的对象是每个元素的对象
                    jQuery(this).wrapAll(html);
                });
    },
```

比较三个方法的不同之处。Wrap 是对每一个元素都进行 wrapAll 的操作，也

就是把每一个元素都包裹在给定的标签中。wrapInner 则是把其每个元素的所有的子元素们进行 wrapAll 的操作。尽管元素的子元素有多个，但是连续的。在子元素的第一个位置之前插入传入的参数标签，之后把这些连续的子元素全部都移到参数标签的内部。可以看作是一种 wrap。但是对于不连续的 wrapALL 就不一定，一般使用时，建议只要采用 wrap 和 wrapInner 就可以了。

5.4dom 元素的内容

Dom 元素提供 innserHTML 来设定元素的内部的 html 内容。这可以直接把 string 的 HTML 变成 Dom 元素，同时也可以把 Dom 元素变成 html 标签的 string. 有的时间我们只需要其 text 内容，不要标签。这个时候可以采用 regexp 把标签给过滤了，就是 text 的内容。

第三种的内容是直接显示出来的 Dom 元素。一个元素的这种内容就是所有子元素。第四种内容指是的元素的 value 值，特别是 Input 类型的元素。因为对于 Input 类型来讲，value 就是其元素内部的 content。

Jquery 也分别就这四种形式提供了实现。

5.4.1 html

Html 是取内部的 html，包含标签，如 xx ,html 返回就是xx。Html 简单，取值就直接 innerHTML,设值就直接插入 value。

```
// 设置每一个匹配元素的html内容。这个函数不能用于XML文档。但可以用于XHTML文档。
// 取得第一个匹配的元素的html内容
   html : function(value) {
       return value == undefined ? (this[0] ? this[0].innerHTML : null) :
this.empty().append(value);// 去掉所有子节点，在内部加上value(字符或dom)
   },
```

在这里还是觉得直接采用 innerHTML,它的速度会更快。只要加上一个判断，如果 value 是 string 的话，就直接采用 innerHTML。不需要采用 append 转来转去。

5.4.2 text

Jquery 的 text 可以把 document.createTextNode(text)的节点加到已经除去所有子元素的当前元素的内部。觉得对于 IE，采用 insertAdjacentText 速度会更快。取值话，text 会把当前元素的所有文本节点都连串成字符串。

```
    // text()
```

```
    // 取得所有匹配元素的内容。结果是由所有匹配元素包含的文本内容组合起来的文本。
    // 对HTML和XML文档都有效。
    // text(val)
    // 设置所有匹配元素的文本内容.与 html() 类似, 但将编码 HTML
    //  "<" 和 ">" 替换成相应的HTML实体).
    text : function(text) {
        if (typeof text != "object" && text != null)//设值
            return this.empty()// 除去元素中所有的子元素，加上创建的文本节点
                    .append((this[0] && this[0].ownerDocument || document)
                createTextNode(text));
        //取值,这里可以看text可以是对象，如是对象，
        //对所有属性对应的元素（一定要是dom元素(节点））进行取值。
        var ret = "";
        jQuery.each(text || this, function() {
             // 所有匹配元素包含的文本内容组合起来
               jQuery.each(this.childNodes, function() {
                   if (this.nodeType != 8)// 8: 注释
                           ret += (this.nodeType != 1// 元素的话，递归子元素
                               ? this.nodeValue: jQuery.fn.text([this]));
                       });
               });
        return ret;
    },
```

5.4.3 value

```
// 获得第一个匹配元素的当前值。
// 在 jQuery 1.2 中,可以返回任意元素的值了。包括select。如果多选，将返回一个数组，其包含所选的值。
// 设置每一个匹配元素的值。在 jQuery 1.2, 这也可以为select元件赋值
val : function(value) {
    if (value == undefined) {//取值
        if (this.length) {//如果当前jquery对象不是空集合
            var elem = this[0];
          //对select中option，我们会采用<option>xx</option>
          //<option value=xx/>的形式
            if (jQuery.nodeName(elem, 'option')) //value属性是否选中
                return (elem.attributes.value || {}).specified
                    ? elem.value: elem.text;
            //处理select的value
            if (jQuery.nodeName(elem, "select")) {
                var index = elem.selectedIndex, values = [],
                  options = elem.options, one = elem.type == "select-one";
```

```
                if (index < 0)  return null;//没有选中选择项
                // 不管是单选还是多选，找到所有的选中的选项
                //对于单选而言，这样的判断实现减少for的次数，提高效率
                //一般的实现都是直接采用for循环，可见jquery代码之优化
                for (var i = one ? index : 0, max = one
                          ? index + 1    : options.length;i < max; i++) {
                    var option = options[i];
                    if (option.selected) {
                      // 嵌套调用本函数，调用其option的部分进行处理，
                      //找到option的指定的值
                        value = jQuery(option).val();
                            if (one) return value;   //单选直接返回值
                            values.push(value);//多选返回数组
                        }
                    }
                    return values;
                } else//不要进行特殊的处理，但要去回车符
                    return (this[0].value || "").replace(/\r/g, "");
            }
           //空集合时返回
            return undefined;
        }
if (value.constructor == Number)value += '';//转换成字符的形式

return this.each(function() {
        if (this.nodeType != 1) return;//不是元素不处理。
        //如果元素是radio|checkbox，设定的值为Array的形式
        //那么本元素的value值或name值在数组中，就设定为选中。
        if (value.constructor == Array
                && /radio|checkbox/.test(this.type))
        this.checked = (jQuery.inArray(this.value, value) >= 0 || jQuery
                        .inArray(this.name, value) >= 0);

        //如果元素是select，先把value转换成数组
        //再判断其options的value在这个数组中不，在的话，就设定为选中。
        else if (jQuery.nodeName(this, "select")) {
            var values = jQuery.makeArray(value);
            jQuery("option", this).each(function() {//给option设值
            this.selected = (jQuery.inArray(this.value,
                values) >= 0 || jQuery.inArray( this.text, values) >= 0);
            });
             if (!values.length) this.selectedIndex = -1;//如果空值
        } else //其他的不要特殊处理，直接设值。
            this.value = value;
```

```
        });
},
```

5.4.4 content

Jquery 的 Content 其实就是元素的所有的子元素的集合。

```
jQuery.each( {
        contents : function(elem) {// iframe?就是文档，或者所有子节点
            return jQuery.nodeName(elem, "iframe") ? elem.contentDocument
                    || elem.contentWindow.document : jQuery
                    .makeArray(elem.childNodes);
        }
    }, function(name, fn) {// 注册到jQuery对象中去，可以调用同名方法
            jQuery.fn[name] = function(selector) {
                var ret = jQuery.map(this, fn);// 每个元素都执行同名方法
                if (selector && typeof selector == "string")
                    ret = jQuery.multiFilter(selector, ret);// 过滤元素集
                return this.pushStack(jQuery.unique(ret));// 构建jQuery对象
                };
        });
```

6.Event 分析

对于 javascript 事件扩展，所有的 lib 都差不多。和 jquery 和 prototype,yui 和 Ext，其要解决的首要问题是兼容性，所有 lib 都会对 event 进行包裹，统一其属性解决其兼容性。对于事件的操作无非是 addEvent,fireEvent,removeEvent 这三个事件方法。一般 lib 都会对浏览器的提供的函数做一些扩展，解决兼容性内存泄漏等问题。第三个问题就是如何得到 domReady 的状态。

6.1 event 的包裹

浏览器的事件兼容性是一个令人头疼的问题。IE 的 event 在是在全局的 window 下，而 mozilla 的 event 是事件源参数传入到回调函数中。还有很多的事件处理方式也一样。

Jquery 提供了一个 event 的包裹，这个相对于其它的 lib 提供的有点简单，但是足够使用。

```
  //对事件进行包裹。
    fix : function(event) {
        if (event[expando] == true) return event;//表明事件已经包裹过
```

```
//保存原始event,同时clone一个。
var originalEvent = event;                                    ①
event = {  originalEvent : originalEvent};
for (var i = this.props.length, prop;i;) {
    prop = this.props[--i];
    event[prop] = originalEvent[prop];
}
event[expando] = true;
//加上preventDefault and stopPropagation，在clone不会运行
event.preventDefault = function() {                           ②
    // 在原始事件上运行
    if (originalEvent.preventDefault)
        originalEvent.preventDefault();
    originalEvent.returnValue = false;
};
event.stopPropagation = function() {
    // 在原始事件上运行
    if (originalEvent.stopPropagation)
        originalEvent.stopPropagation();
    originalEvent.cancelBubble = true;
};
// 修正 timeStamp
event.timeStamp = event.timeStamp || now();
// 修正target
if (!event.target)                                            ③
    event.target = event.srcElement || document;
if (event.target.nodeType == 3)//文本节点是父节点。
    event.target = event.target.parentNode;
// relatedTarget
if (!event.relatedTarget && event.fromElement)                ④
    event.relatedTarget = event.fromElement == event.target
            ? event.toElement : event.fromElement;
// Calculate pageX/Y if missing and clientX/Y available
if (event.pageX == null && event.clientX != null) {           ⑥
    var doc = document.documentElement, body = document.body;
  event.pageX = event.clientX
        + (doc && doc.scrollLeft || body && body.scrollLeft || 0)
            - (doc.clientLeft || 0);
    event.pageY = event.clientY
        + (doc && doc.scrollTop || body && body.scrollTop || 0)
            - (doc.clientTop || 0);
}

// Add which for key events
```

```
if (!event.which  && ((event.charCode || event.charCode === 0) ⑦
                 ? event.charCode  : event.keyCode))
        event.which = event.charCode || event.keyCode;

// Add metaKey to non-Mac browsers
    if (!event.metaKey && event.ctrlKey)                    ⑧
        event.metaKey = event.ctrlKey;
// Add which for click: 1 == left; 2 == middle; 3 == right
// Note: button is not normalized, so don't use it
    if (!event.which && event.button)                      ⑨
        event.which = (event.button & 1 ? 1 : (event.button & 2
               ? 3 : (event.button & 4 ? 2 : 0)));
    return event;
},
```

上面的代码①处保留原始事件的引用，同时 clone 原始事件。在这个 clone 的事件上进行包裹。②处在原始事件上运行 preventDefault 和 stopPropagation 两个方法达到是否阻止默认的事件动作发生和是否停止冒泡事件事件向上传递。

③处是修正 target 个，IE 中采用 srcElement，同时对于文本节点事件，应该把 target 传到其父节点。

④处relatedTarget只是对于mouseout、mouseover有用。在IE中分成了to和from两个Target变量，在mozilla中没有分开。为了保证兼容，采用relatedTarget统一起来。

⑥处是进行 event 的坐标位置。这个是相对于 page。如果页面可以 scroll，则要在其 client 上加上 scroll。在 IE 中还应该减去默认的 2px 的 body 的边框。

⑦处是把键盘事件的按键统一到 event.which 的属性上。Ext 中的实现 ev.charCode || ev.keyCode || 0; ⑨则是把鼠标事件的按键统一把 event.which 上。charCode、ev.keyCode 一个是字符的按键，一个不是字符的按键。⑨处采用&的方式来进行兼容性的处理。 Ext 通过下面三行解决兼容问题。

var btnMap = Ext.isIE ? {1:0,4:1,2:2} : (Ext.isSafari ? {1:0,2:1,3:2} : {0:0,1:1,2:2}); this.button = e.button ? btnMap[e.button] : (e.which ? e.which-1 : -1);

①②③④⑤⑥⑦⑧⑨⑩

6.2 事件的处理

Jquery 提供了一些来进行 regist,remove,fire 事件的方法。

6.2.1 Register

对于注册事件，jquery 提供了 bind、one、toggle、hover 四种注册事件的方法，bind 是最基本的方法。One 是注册只运行一次的方法，toggle 注册交替运行的方法。Hover 是注册鼠标浮过的方法。

```
bind : function(type, data, fn) {
    return type == "unload" ? this.one(type, data, fn) : this
        .each(function() {// fn || data, fn && data实现了data参数可有可无
                jQuery.event.add(this, type, fn || data, fn && data);
            }); },
```

Bind 中对于 unload 的事件，只能运行一次，其它的就采用默认的注册方式。

```
// 为每一个匹配元素的特定事件（像click）绑定一个一次性的事件处理函数。
// 在每个对象上，这个事件处理函数只会被执行一次。其他规则与bind()函数相同。
// 这个事件处理函数会接收到一个事件对象，可以通过它来阻止（浏览器）默认的行为。
// 如果既想取消默认的行为，又想阻止事件起泡，这个事件处理函数必须返回false。
    one : function(type, data, fn) {
      var one = jQuery.event.proxy(fn || data, function(event) {
            jQuery(this).unbind(event, one);
            return (fn || data).apply(this, arguments);/this->当前的元素
                });
            return this.each(function() {
                jQuery.event.add(this, type, one, fn && data);
            });

        },
```

One 与 bind 基本上差不多，不同的在调用 jQuery.event.add 时，把注册的事件处理的函数做了一个小小的调整。One 调用了 jQuery.event.proxy 进行了代理传入的事件处理函数。在事件触发调用这个代理的函数时，先把事件从 cache 中删除，再执行注册的事件函数。这里就是闭包的应用，通过闭包得到 fn 注册的事件函数的引用。

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

        },
```

Hover 则是建立在 bind 的基础之上。

```
//每次点击后依次调用函数。
toggle : function(fn) {
  var args = arguments, i = 1;
  while (i < args.length)//每个函数分配GUID
        jQuery.event.proxy(fn, args[i++]);//修改后的还在args中
 return this.click(jQuery.event.proxy(fn, function(event) {//分配GUID
        this.lastToggle = (this.lastToggle || 0) % i;//上一个函数
        event.preventDefault();//阻止缺省动作
        //执行参数中的第几个函数，apply可以采用array-like的参数
        return args[this.lastToggle++].apply(this,arguments)||false;
    }));
    },
```

Toggle 中参数可以是多个 fn。先把它们代码生成 UUID。之后调用 click 的方法来注册再次进行代理的 callback。这个函数在事件触发时运行，它先计算上一次是执行了参数中的那个函数。之后阻止缺省动作。之后找到下一个函数运行。

```
//为jquery对象增加常用的事件方法
jQuery.each(
    ("blur,focus,load,resize,scroll,unload,click,dblclick,"
    + "mousedown,mouseup,mousemove,mouseover,mouseout,change,select,"
    + "submit,keydown,keypress,keyup,error").split(","),
function(i, name) {jQuery.fn[name] = function(fn) {
                return fn ? this.bind(name, fn) : this.trigger(name);
            };});
```

Jquery 增加了一个常用的事件处理方法，包含上面调用的 click。这里可以看出这里还是调用 bind 进行注册。当然这里还可以通过程序实现去触发事件。

上面的众多方法都是注册事件，其最终都落在 jQuery.event.add();来完成注册的功能。如果我们采用 Dom0 或 DOM1 的事件方法，我们会采用 elem.onclick=function(){}来为元素的某一种事件来注册处理函数。这个最大的缺点就是每个一个事件只是一个处理函数。在 dom1 的方式中有改进，我们可以采用 elem.addEventListener(type, handle, false)为元素的事件注册多个处理函数。

这样的处理方式还不是很完美，如果我们只这个事件运行一次就有点麻烦了。我们要在事件的处理函数中最后进行 elem.removeEventListener 来取消事件的监听。这样做可能会有事务上的问题。如果第一个事件处理函数在没有取消事件监听之前，就再次触发了怎么办？

还有采用浏览器的方式，它不支持自定义事件的注册和处理，还不能为多个事件注册同一个处理函数。

```
jQuery.event = {// add 事件到一个元素上。
```

```
add : function(elem, types, handler, data) {
 if (elem.nodeType == 3 || elem.nodeType == 8) return;// 空白节点或注释
    // IE不能传入window,先复制一下。
if (jQuery.browser.msie && elem.setInterval) elem = window;
// 为handler分配一个全局唯一的Id
if (!handler.guid)   handler.guid = this.guid++;
// 把data附到handler.data中
if (data != undefined) {                                          ①
 var fn = handler;
 handler =this.proxy(fn,function(){return fn.apply(this,arguments);});
 handler.data = data;
    }
// 初始化元素的events。如果没有取到events中值，就初始化data: {}      ②
var events =jQuery.data(elem,"events")||jQuery.data(elem,"events",{}),
// 如果没有取到handle中值，就初始化data: function() {....}        ③
handle = jQuery.data(elem, "handle")|| jQuery.data(elem, "handle",
   function() {//处理一个触发器的第二个事件和当page已经unload之后调用一个事件。
        if (typeof jQuery != "undefined"&& !jQuery.event.triggered)
            return jQuery.event.handle.apply(//callee.elem=handle.elem
                  arguments.callee.elem, arguments);
            });
// 增加elem做为handle属性，防止IE由于没有本地Event而内存泄露。
 handle.elem = elem;
// 处理采用空格分隔多个事件名，如jQuery(...).bind("mouseover mouseout", fn);
jQuery.each(types.split(/\s+/), function(index, type) {      ④
    // 命名空间的事件，一般不会用到。
  var parts = type.split(".");type = parts[0];handler.type = parts[1];
    // 捆绑到本元素type事件的所有处理函数
 var handlers = events[type];                                      ⑤
 if (!handlers) {// 没有找到处理函数列表就初始化事件队列
        handlers = events[type] = {};
    // 如果type不是ready，或ready的setup执行返回false                ⑥
  if (!jQuery.event.special[type]|| jQuery.event.special[type].setup
        .call(elem, data) === false) {// 调用系统的事件函数来注册事件
   if(elem.addEventListener)elem.addEventListener(type,handle,false);
     else if (elem.attachEvent)elem.attachEvent("on" + type, handle);
          }
      }
// 把处理器的id和handler形式属性对的形式保存在handlers列表中，
// 也存在events[type][handler.guid]中。
handlers[handler.guid] = handler;                                 ⑦
// 全局缓存这个事件的使用标识
jQuery.event.global[type] = true;
});
```

```
elem = null; // 防止IE内存泄露。
},
guid : 1,
global : {},
```

jQuery.event.add 通过 jQuery.data 把事件相关的事件名和处理函数有机有序地组合起存放在 jQuery.cache 中与该元素对应的空间里。我们就一个例子分析一下 add 的过程中：假如我们招待下面 jQuery(e1).bind("mouseover mouseout", fn0);jQuery(e1).bind("mouseover mouseout", fn1)的语句。

在 jQuery(e1).bind("mouseover mouseout", fn0);时，②③都不可能从 cache 取到数，先初始化。此时的 `cache:{e1_uuid:{events:{},handle:fn}}`。接着在⑤会为 mouseover mouseout 名初始化。此时的 cache: `{e1_uuid:{events:{ mouseover:{}, mouseout:{}},handle:fn}}`。在⑥处向浏览器的事件中注册处理函数。接着⑦会把处理函数到事件名中。此时的 `cache:{e1_uuid:{events:{mouseover:{fn0_uuid:fn0},mouseout:{ fn0_uuid:fn0}},handle:fn}}`。这里可以看出为采用 proxy 为函数生成 uuid 的作用了。

在 jQuery(e1).bind("mouseover mouseout", fn1)时，②③都从 cache 取到数据 `{e1_uuid:{events:{mouseover:{fn0_uuid:fn0},mouseout:{ fn0_uuid:fn0}},` 接着在⑤取到 `mouseover:{fn0_uuid:fn0},mouseout:{ fn0_uuid:fn0}`的引用。接着⑦会把处理函数注册到事件名中。此时的 `cache:{e1_uuid:{events:{mouseover:{fn0_uuid:fn0,fn1_uuid:fn1,},mouseout:{ fn0_uuid:fn0, fn1_uuid:fn1}},handle:fn}}`。

jQuery.event.add 很重要的任务就是把注册的事件函数有序地存放起来。以便 remove 和 fire 事件的函数能找到。

```
//{elem_uuid_1:{events:{mouseover:{fn_uuid:fn1,fn_uuid1:fn2},
            //mouseout:{fn_uuid:fn1,fn_uuid1:fn2}},handle:fn}}
```

6.2.2 trigger

注册了事件，如 onclick。那么当用户点击这个元素时，就会自动触发这个事件的已经注册的事件处理函数。但是我们有的时候要采用程序来模拟事件的触发就得采用强迫触发某个事件。在 IE 中我们可以采用.fireEvent()来实现。如：<form onsubmit="a()" >中，如果 button 的 form.submit()的方式提交表单，是不会主动触发 onsumbit 事件的，如果必须的话，就要在 submit 前 $(“:form”)[0].fireEvent("onsubmit”,)，这样就会触发该事件。

在 mozilla 中有三个步骤： var evt = document.createEvent('HTMLEvents'); evt.initEvent('change',true,true); t.dispatchEvent(evt);
在 prototype 是采用这样的方式来实现的。那么 jquery 中呢，它的实现方式有一点不一样。

```
trigger : function(type, data, fn) {
  return this.each(function() {
      jQuery.event.trigger(type, data, this, true, fn);
        }); },
```

Trigger 有三个参数，data 参数是为了注册的事件函数提供了实传。如果 data[0]中 preventDefault 存在，data[0]就可以做为用户自定义的包裹事件的空间。Fn 是可以为事件提供一个即时即用的事件处理方法。也就是在没有注册事件的情况下也可以通过传入处理函数来处理事件。如果已经注册了，那就是在原来的事件处理函数之后执行。

```
   //这个方法将会触发指定的事件类型上所有绑定的处理函数。但不会执行浏览器默认动作.
triggerHandler : function(type, data, fn) {
    return this[0]&& jQuery.event.trigger(type,data,this[0],false,fn);
   },
```

triggerHandle 通过把 jQuery.event.trigger 的 donative 参数设为 false，来阻止执行浏览器默处理方法。它与 trigger 不现的一点，还在于它只是处理 jquery 对象的第一个元素。

上面两个方法都调用了 jQuery.event.trigger 来完成任务：

```
trigger : function(type, data, elem, donative, extra) {
   data = jQuery.makeArray(data);//data可以为{xx:yy}
   //支持getData!这样的形式，exclusive = true表现会对add的注册的
   //事件的所有函数进行命名空间的分种类的来执行。
 if (type.indexOf("!") >= 0) {                    ①
      type = type.slice(0, -1);var exclusive = true;
      }
 if (!elem) {// 处理全局的fire事件                  ②
   if (this.global[type])
      jQuery.each(jQuery.cache, function() {
        // 从cache中找到所有注册该事件的元素，触发改事件的处理函数
          if (this.events && this.events[type])
              jQuery.event.trigger(type, data, this.handle.elem);
          });
   } else {// 处理单个元素事件的fire事件                ③
    if (elem.nodeType == 3 || elem.nodeType == 8)    return undefined;
     var val, ret, fn = jQuery.isFunction(elem[type] || null),
    // 如果data参数传进入的不是浏览器的event对象的话，event变量为true.
   //如果data参数本身是娄组，那么第一个元素不是浏览器的event对象时为true.
   //对于event为true。即没有event传进入，先构建一个伪造的event对象存在data[0]。
```

```
    event = !data[0] || !data[0].preventDefault;
    // 在没有传入event对象的情况下，构建伪造event对象。
    if (event) {//存到数组中的第一个                          ④
        data.unshift( { type : type,target : elem,
            preventDefault : function() {},stopPropagation :
            function() {}, timeStamp : now()   });
       data[0][expando] = true; // 不需要修正伪造的event对象
        }
     data[0].type = type; //防止事件名出错
    //表现会进行事件注册函数的分类（命名空间）执行。不是所有的。
     if (exclusive) data[0].exclusive = true;

    //与prototype等传统的处理方式不一样，没有采用fireEvent来
    //来fire通过注册到浏览器事件中的事件处理方法。
    //这里分了三步，先fire通过jQuery.event.add来注册的事件，这个事件
    //有可能是自定义的事件（没有注册到浏览器事件中）。
    //第二步是fire通过elem.onclick方式注册的事件的本地处理函数
     //第三步是fire默认的事件处理方式（在本地的onclick的方式注册
     //不存在的情况下）。

     // 这里是触发通过jQuery.event.add来注册的事件，
      var handle = jQuery.data(elem, "handle");          ⑤
      if (handle)val = handle.apply(elem, data); //这里data分成多个参数
    //处理触发通过elem.onfoo=function()这样的注册本地处理方法，
    //但是是对于links 's .click()不触发,这个不会执行通过addEvent
    //方式注册的事件处理方式。
    if ((!fn || (jQuery.nodeName(elem, 'a') && type == "click")) ⑥
        && elem["on"+ type]&& elem["on"+type].apply(elem,data) === false)
      val = false;
  //额外的函数参数的开始几个是通过data给定的。这里会把伪造加上的event给去掉。
  //它的最后一个参数是一系列的事件处理函数返回的结果，一般为bool值
  //这个函数可以根据这个结果来处理一个扫尾的工作。
    if (event) data.shift();
  // 处理触发extra给定的函数处理。
    if (extra && jQuery.isFunction(extra)) {                  ⑦
         ret = extra.apply(elem, val == null ? data : data.concat(val));
         //如果这个函数有返回值，那么trigger的返回值就是它的返回值
         //没有的话就是串连的事件处理函数的最后一个返回值。一般为bool
        if (ret !== undefined)  val = ret;
     }
    // 触发默认本地事件方法，它是在没有如.onclick注册事件
    //加上前面的执行事件处理函数返回值都不为false的情况下，才会执行。
    //它还可以通donative来控制是否执行。
    //如form中可以采用this.submit()来提交form.
```

```
    if (fn && donative !== false && val !== false          ⑧
            && !(jQuery.nodeName(elem, 'a') && type == "click")) {
        this.triggered = true;
        try {elem[type]();   //对于一些hidden的元素，IE会报错
            } catch (e) {}
        }
    this.triggered = false;
    }
return val;
},
```

Jquery 的 fire 事件的方法与 prototype 中实现是完全不一样的。Ext、YUI 没有提供强迫触发事件的方法。对于一般的思维，程序来触发浏览器的事件就应该采用 fireEvent 或 dispatchEvent 方法来运行。

但是 jquery 采用一种不同的方法。对于通过 jquery.event.add 来注册的事件（不管是自定义的还是注册到浏览器事件），它保存在一个与元素及事件名相对应的 cache 中。在浏览器的触发中，这个是没有什么作用。但是它是为了通过等程序来强迫触发时，从 cache 中取到对应的事件处理函数。这个时候就抛开了浏览器的事件。在这里还可以执行一些自定义的事件函数。如⑤处。

对于通过 html 的标签中如 click 或 elem.onclick=function(){}形式注册的事件函数。在⑥处它采用执行元素的如 onclick 形式的回调函数就可以。通过这种 dom0 的方式只能注册一个函数。

有的时候，如果没有 onclick 这样的事件处理函数，浏览器会执行默认的处理函数。如 form.submit()。⑧处可以看出对于这样的默认的事件处理，还可以通过参数 donative 来控制。

程序手动强迫触发事件，有一点问题就是 event 是怎么生成，就是没有浏览器生成 event 传入到函数中。Prototype 采用了是新生成的 dataavailable 的事件。这样的事件也没有什么作用。Jquery 也采用 fake 的方式伪造一个一个事件，如④，它比 prototype 的事件好处在于它能通过 trigger 的函数的参数来传入需要的 event。Prototype 则不能。

通过上面的分析，隐隐可以看出 Jquery 是通过模拟浏览器的触发事件的执行过程来构建这个 trigger 的函数的。先执行 dom1 方式（addEvent）注册的事件，再执行 dom0 方式注册的事件，最后看看要不要执行默认的事件处理。

在⑦处，我们可以看出 trigger 还可能通过传入回调函数和参数来完成对执行的事件处理函数的结果进行判断处理，形成新结果通过 trigger 的函数返回。这在有的时候是很有用的。

除了这些，它还能对于事件的处理函数进行分类（namespace），可以在合适的时候调用事件的不同分类的的处理函数（通过 jquery.event.add 来注册）。这个分

类的处理在 handle 实现。

```
handle : function(event) {
    // 返回 undefined or false
    var val, ret, namespace, all, handlers;
   //修改了传入的参数，这里是引用。
    event = arguments[0] = jQuery.event.fix(event || window.event);
    // 命名空间处理
    namespace = event.type.split(".");
    event.type = namespace[0];
    namespace = namespace[1];
    // all = true 表明任何 handler,namespace不存在，同时
    //event.exclusive不存在或为假时，all=true.
    all = !namespace && !event.exclusive;
    // 找到元素的events中缓存的事件名的处理函数列表
    handlers = (jQuery.data(this, "events") || {})[event.type];
    for (var j in handlers) {// 每个处理函数执行
        var handler = handlers[j];
        // Filter the functions by class
        if (all || handler.type == namespace) {
            // 传入引用，为了之后删除它们
            event.handler = handler;
            event.data = handler.data;//add的时候加上的
            ret = handler.apply(this, arguments);// 执行事件处理函数
            if (val !== false)
               val = ret;// 只要有一个处理函数返回false，本函数就返回false.
            if (ret === false) {// 不执行浏览器默认的动作
                event.preventDefault();
                event.stopPropagation();
            }
        }
    }
    return val;   },
```

handle 的主要功能是就是分类且有序地执行事件的所有的注册的处理函数。

6.2.3 remove

Remove 就是除去事件的监听，在这里的实现，它还要除去 cache 中对应的的数据。

```
// 从元素中remove一个事件
remove : function(elem, types, handler) {
    if (elem.nodeType == 3 || elem.nodeType == 8) return;
    // 取出元素的events中Fn列表
```

```
    var events = jQuery.data(elem, "events"), ret, index;
    if (events) {
            // remove所有的该元素的事件 .是命名空间的处理
        if (types == undefined
                || (typeof types == "string" && types.charAt(0) == "."))
             for (var type in events)
                this.remove(elem, type + (types || ""));//xx.yy
        else {
               // types, handler参数采用{type:xxx,handler:yyy}形式
            if (types.type) {
                handler = types.handler;
                types = types.type;
               }
// 处理采用空格分隔多个事件名 jQuery(...).unbind("mouseover mouseout", fn);
    jQuery.each(types.split(/\s+/), function(index, type) {
                // 命名空间的事件，一般不会用到。
        var parts = type.split(".");
        type = parts[0];
        if (events[type]) {// 事件名找到
          if (handler)// handler传入，就remove事件名的这个处理函数
            delete events[type][handler.guid];//guid的作用
          else // remove这个事件的所有处理函数，带有命名空间的处理
             for (handler in events[type])
               if (!parts[1]|| events[type][handler].type == parts[1])
                  delete events[type][handler];
        // 如果没有该事件的处理函数存在，就remove事件名
            for (ret in events[type])   break; // 看看有没有?
             if (!ret) {// 没有
               if (!jQuery.event.special[type] || jQuery.event.special
                [type].teardown.call(elem) === false) {//type不等于ready
                  if (elem.removeEventListener)// 在浏览器中remove事件名
                   elem.removeEventListener(type,jQuery.data(elem,
                                    handle"),  false);
                  else if (elem.detachEvent)
                     elem.detachEvent("on" + type,jQuery.data(elem,
                                    "handle"));
               }
            ret = null;
            delete events[type];// 在缓存中除去。
               }
            }
    });
}
```

```
// 不再使用，除去expando
for (ret in events)     break;
    if (!ret) {
            var handle = jQuery.data(elem, "handle");
            if (handle)
                handle.elem = null;
            jQuery.removeData(elem, "events");
            jQuery.removeData(elem, "handle");
        }
    }
},
```

①②③④⑤⑥⑥⑦⑧⑨

6.3 domReady 的处理

Domready 是每个 lib 都要实现的函数，因为 dom 还没有完全 ready，那么对于对于 dom 元素的操作可能出错，因为 dom 树中可能还没有 Load 这个元素。为了保障不出现这样的错误，就出现地 domready。对于每种浏览器 Domready 都有着自己不同的判断。

Jquery 的 domready 的实现和其它的 lib 的实现没有什么区别。

```
//dom ready时执行 fn
ready : function(fn) {
        bindReady();//注册监听
        if (jQuery.isReady)//ready就运行
            fn.call(document, jQuery);
        else
            // 增加这个函数到queue中。可见支持无数的ready的调用。
            jQuery.readyList.push(function() {
                return fn.call(this, jQuery);
            });
        return this;
    }
```

我们通过$(fn)的方法就是在调用这个方法。它首先通过 bindReady()来注册监听 dom 是否已经 loaded。如果已经 loaded 就运行 fn。如果没有就把 fn 存放在 readyList 中。对于多个$(fn)，把它们各自的 fn 参数保存在公共的 jQuery. readyList 的公共集合中。待到 dom loaed 之后统一运行。对于 dom 已经 loaded，就可以分别去运行 fn。

```
var readyBound = false;
```

```
function bindReady() {
    if (readyBound)
        return;
    readyBound = true;

    // Mozilla, Opera, webkit nightlies 支持DOMContentLoaded事件
if (document.addEventListener && !jQuery.browser.opera)
        //当DOMContentLoaded事件触发时就运行jQuery.ready
document.addEventListener("DOMContentLoaded", jQuery.ready, false);

    //IE或不是frame的window
    if (jQuery.browser.msie && window == top)
        (function() {
            if (jQuery.isReady)
                return;
            try {
                // 在ondocumentready之前，一直都会抛出异常
                // http://javascript.nwbox.com/IEContentLoaded/
                document.documentElement.doScroll("left");
            } catch (error) {
                //一直运行bindReady()(=arguments.callee)
                setTimeout(arguments.callee, 0);
                return;
            }
            jQuery.ready();//documentready就运行jQuery.ready
        })();

    if (jQuery.browser.opera)
        document.addEventListener("DOMContentLoaded", function() {
            if (jQuery.isReady)
                return;
                //只有styleSheets完全enable时，才是完全的load,其实还有pic
        for (var i = 0;i < document.styleSheets.length; i++)
          if (document.styleSheets[i].disabled) {//通过styleSheets来判断
                    setTimeout(arguments.callee, 0);
                    return;
                }
                jQuery.ready();
            }, false);

    if (jQuery.browser.safari) {
        var numStyles;
        (function() {
            if (jQuery.isReady)
```

```
                return;
                //首先得得到readyState=loaded或=complete
            if (document.readyState != "loaded"
                    && document.readyState != "complete") {
                setTimeout(arguments.callee, 0);
                return;
            }
            //取得style的length,比较它们之间的长度，看看是不是完成loaded
            if (numStyles === undefined)
                numStyles = jQuery("style,
                   link[rel=stylesheet]").length;
            if (document.styleSheets.length != numStyles) {
                setTimeout(arguments.callee, 0);
                return;
            }
            jQuery.ready();
        })();
    }

    //最后只能依赖于window.load.
    jQuery.event.add(window, "load", jQuery.ready);
}
```

这段代码看起来很多，其实就是采用 setTimeout(arguments.callee, 0);反复来运行 bindReady。如果其得到 dom ready 的条件满足的话，就执行 jQuery.ready()来执行通过$(fn)注册的 fn 函数。对于每种浏览器，这个满足的条件是不一样。上面的代码就是针对于几种常用的浏览器分别做了各自的处理。

```
  isReady : false,
    readyList : [],
    // Handle when the DOM is ready
        ready : function() {
            if (!jQuery.isReady) {
                jQuery.isReady = true;
                if (jQuery.readyList) {
                    jQuery.each(jQuery.readyList, function() {
                        this.call(document);
                    });
                    jQuery.readyList = null;
                }
                jQuery(document).triggerHandler("ready");
            }
        }

    });
```

当运行到jQuery.ready()的时候就说明dom已经完全的Loaded，那么现在就应该执行

保存在jQuery.readyList中的fn。jQuery.ready()就是完成这个工作。

7.Ajax 分析

7.1 jquery 的 ajax 常用方法

对于 Ajax 原理不深入分析。Jquery 肯定也会提供 Ajax 的实现。对于 ajax 的请求，可以分成如下的几步：

1、通过 new XMLHttpRequest 或其它的形式（指 IE）生成 ajax 的对象 xhr。

2、通过 xhr.open(type, url, async, username, password)的形式建立一个连接。

3、通过 setRequestHeader 设定 xhr 的请求头部（request header）。

4、通过 send(data)请求服务器端的数据。

5、执行在 xhr 上注册的 onreadystatechange 回调处理返回数据。

任何的 lib 都是在这几步之上进行相关扩展而达到自己的功能。这几步之中，对于 url，我们要考虑是跨域请求怎么办，ajax 为了安全的考虑不支持跨域请求，那么对于这个问题，Jquery 和 Ext 都是采用 scriptTag 的方式。

Jquery 主要就是解决上面这问题，之后就在这个基础之上进行扩展，如 getScript,getJson 之类的函数。Ajax 一个很重要的任务就是提交 form。Jquery Ajax 提供了如 Prototype 的 form 中 serializeElements 把 form 的元素串行起请求字符串。这是 ajax 的辅助方法。

对于 ajax 的应用，不管对返回的数据进行如何的处理，其最终目的还是得落在页面的显示上，也是某一个或一些 dom 元素上。那么我从这个需要改变内容的 dom 元素（集合）出发，通过 ajax 去获得数据进行一些处理最终填充到给定的元素中。这和 prototype 中 Ajax.Updata 相似。

load

Jquery 对象的 load(url, params, callback)函数就是完成这类似的工作的。

```
// 载入远程 HTML 文件代码并插入至 DOM 中。
// 默认使用 GET 方式 - 传递附加参数时自动转换为 POST 方式。jQuery 1.2 中，
//可以指定选择符，来筛选载入的 HTML 文档，DOM 中将仅插入筛选出的 HTML 代码。
// 语法形如 "url #some > selector"。
load : function(url, params, callback) {
    if (typeof url != 'string')     return this._load(url);
```

```
    var off = url.indexOf(" ");// 找到第一个空格处
    if (off >= 0) {                                                    ①
        var selector = url.slice(off, url.length);// 第一个空格之后的字符
        url = url.slice(0, off);// 第一个空格之前的字符
        }
  callback = callback || function() {
    };
var type = "GET";// 默认的是get类型
// 这里是判断第二参数，如果是fn,那么就是指callback
// 如果是object,那么就是指data.load(url,[data],[callback])
if (params)
    if (jQuery.isFunction(params)) {
        callback = params;   params = null;}
    else if (typeof params == 'object') {
        params = jQuery.param(params);type = "POST";       ②
    }
var self = this;
jQuery.ajax( {//Ajax请求                                           ③
  url : url,   type : type,dataType : "html",data : params,
  complete : function(res, status) {
        // 成功就注射html到所有匹配的元素中
    if (status == "success" || status == "notmodified")
        // selector是否指定？没有的话就是全部的内容
        // 指定的话，就是生成dom文档的形式，之后在中间找到满足条件的元素。
        // 这中间删除 scripts 是避免IE中的 'Permission Denied' 错误
  self.html(selector ? jQuery("<div/>")                    ④
    .append(res.responseText.replace(/<script(.|\s)*?\/script>/g, ""))
            .find(selector) : res.responseText);
 self.each(callback, [res.responseText, status, res]); // 执行回调⑤
    }
});
return this;

        },
```

上面的代码①可以看出 load 的 url 参数可以指定选择符，来筛选载入的 HTML 文档，DOM 中将仅插入筛选出的 HTML 代码。语法形如 "url #some > selector"。在④处通过 html 的方式插入到元素的内部（取代）。其支持的 response 应该是 html 的片断。

param

在②处通过调用了 jQuery.param(params)来完成对 params 的 key/value 的对象

形式转换成查寻字符串的形式。

```
// 串行化form子元素组成的数组或对象形式查询字符串
    param : function(a) {
        var s = [];
        function add(key, value) {
            s[s.length] = encodeURIComponent(key) + '='
                    + encodeURIComponent(value);
        };
/ 对于数组的参数，每个元素({name:xx,value:yy})都串行化为key/value的字符串。
        if (a.constructor == Array || a.jquery)
            jQuery.each(a, function() {
                add(this.name, this.value);
            });
        // 对于对象{a1:{name:xx,value:yy},a2:{name:xx,value:yy}}
        // 都串行化为key/value的字符串。
        else
            for (var j in a)
                // value是数组，key 名字要重复
                if (a[j] && a[j].constructor == Array)
                    jQuery.each(a[j], function() {
                        add(j, this);
                    });
                else
                    add(j, jQuery.isFunction(a[j]) ? a[j]() : a[j]);
        // 返回生成字符串
        return s.join("&").replace(/%20/g, "+");
    }
```

Get&post

Load 有的时候也不能很好地完成功能，如果不是 html 的 response。那么不能采用了。Jquery 还提供了几个静态方式：jquery.get()、jquery. getScript ()、jquery. getJSON ()、jquery. post ()。jquery. getScript ()、jquery. getJSON ()不过是在 jquery.get()基础之上提供了某方面的简单的处理。Get 和 Post 在这里没有什么不一样，除了请求的 type 不一样之外。

```
 //通过get的type方式进行ajax的请求
    get : function(url, data, callback, type) {
        // 前移 arguments 如data 参数省略
        if (jQuery.isFunction(data)) {
            callback = data;  data = null;  }
        return jQuery.ajax( {    type : "GET", url : url,
```

```
            data : data,  success : callback,  dataType : type
        });
    },
  //以post方式进行ajax请求
post : function(url, data, callback, type) {
        if (jQuery.isFunction(data)) {
            callback = data;  data = {}; }
        return jQuery.ajax( {type : "POST",url : url,
            data : data,  success : callback,dataType : type     });
    },
```

Get 和 post 都是在 ajax 的请求方面没有什么区别。除了在服务器的接收处理的方面有所不同之外。如 get 在.net 中是 request.querstring，而 post 则是采用 request.form 来获得。

```
    //取得返回的script
    getScript : function(url, callback) {
        return jQuery.get(url, null, callback, "script");
    },
   //取得json
    getJSON : function(url, data, callback) {
        return jQuery.get(url, data, callback, "json");
    },
```

getScript、getJSON 都是在调用 jQuery.get,传入不同的 datatype 的形式来区别不同的类型。

7.2 jquery.ajax

这所有的最终都是通过 jQuery.ajax()来完成的。

```
 ajax : function(s) {
    //两次继承s,以便在测试中能检测
    s = jQuery.extend(true, s, jQuery.extend(true, {},
                    jQuery.ajaxSettings, s));        ①
    var jsonp, jsre = /=\?(&|$)/g, status, data,
        type = s.type .toUpperCase();
// 如果不是字符集串就转换在查询字符集串
if (s.data && s.processData && typeof s.data != "string")
        s.data = jQuery.param(s.data);

// 构建jsonp请求字符集串。jsonp是跨域请求，要加上callback=? 后面将会加函数名
if (s.dataType == "jsonp") {                          ②
  if (type == "GET") {//使get的url包含 callback=? 后面将会进行加函数名
        if (!s.url.match(jsre))
```

```
            s.url += (s.url.match(/\?/) ? "&" : "?")
                        + (s.jsonp || "callback") + "=?";
            } // 构建新的s.data，使其包含callback=function name
  else if (!s.data || !s.data.match(jsre))
        s.data = (s.data ? s.data + "&" : "") + (s.jsonp||"callback")+ "=?";
 s.dataType = "json";
 }
//判断是否为jsonp,如果是，进行处理。
if (s.dataType == "json"
            && (s.data && s.data.match(jsre) || s.url.match(jsre))) {③
     jsonp = "jsonp" + jsc++;
    / /为请求字符集串的callback=加上生成回调函数名
    if (s.data)s.data = (s.data + "").replace(jsre, "=" + jsonp + "$1");
    s.url = s.url.replace(jsre, "=" + jsonp + "$1");

            // 我们需要保证jsonp 类型响应能正确地执行
    //jsonp的类型必须为script。这样才能执行服务器返回的
    //代码。这里就是调用这个回调函数。
    s.dataType = "script";
   //window下注册一个jsonp回调函数有，让ajax请求返回的代码调用执行它，
    //在服务器端我们生成的代码 如callbackname(data);形式传入data.
    window[jsonp] = function(tmp) {
        data = tmp;success();complete();
    // 垃圾回收,释放联变量，删除jsonp的对象，除去head中加的script元素
       window[jsonp] = undefined;
        try {  delete window[jsonp];
            } catch (e) { }
        if (head)  head.removeChild(script);
        };
    }

if (s.dataType == "script" && s.cache == null)   s.cache = false;
// 加上时间戳，可见加cache:false就会加上时间戳
if (s.cache === false && type == "GET") {
    var ts = now();
    var ret = s.url.replace(/(\?|&)_=.*?(&|$)/, "$1_=" + ts + "$2");
    // 没有代替，就追加在url的尾部
    s.url = ret+ ((ret == s.url) ? (s.url.match(/\?/) ? "&" : "?") + "_="
                      + ts : "");
    }
// data有效，追加到get类型的url上去
if (s.data && type == "GET") {
        s.url += (s.url.match(/\?/) ? "&" : "?") + s.data;
        // 防止IE会重复发送get和post data
```

```
        s.data = null;
        }
// 监听一个新的请求
if (s.global && !jQuery.active++) jQuery.event.trigger("ajaxStart");④
// 监听一个绝对的url,和保存domain
 var parts = /^(\w+:)?\/\/([^\/?#]+)/.exec(s.url);
// 如果我们正在请求一个远程文档和正在load json或script通过get类型
//location是window的属性，通过和地址栏中的地址比较判断是不是跨域。
if (s.dataType == "script"  && type == "GET"&& parts && (parts[1] &&
  parts[1] != location.protocol || parts[2] != location.host)) {⑤
        // 在head中加上<script src=""></script>
        var head = document.getElementsByTagName("head")[0];
        var script = document.createElement("script");
        script.src = s.url;
        if (s.scriptCharset) script.charset = s.scriptCharset;
        //如果datatype不是jsonp，但是url却是跨域的。采用scriptr的
        //onload或onreadystatechange事件来触发回调函数。
       if (!jsonp) {
            var done = false;
            // 对所有浏览器都加上处理器
            script.onload = script.onreadystatechange = function() {
            if (!done&& (!this.readyState || this.readyState == "loaded"
                  || this.readyState == "complete")) {
                    done = true;  success();
                    complete();head.removeChild(script);
                }
        };
    }
    head.appendChild(script);
   // 已经使用了script 元素注射来处理所有的事情
    return undefined;
}
var requestDone = false;
// 创建request,IE7不能通过XMLHttpRequest来完成，只能通过ActiveXObject
var xhr = window.ActiveXObject                                  ⑥
        ? new ActiveXObject("Microsoft.XMLHTTP"): new XMLHttpRequest();
// 创建一个请求的连接，在opera中如果用户名为null会弹出login窗口中。
if (s.username)xhr.open(type, s.url, s.async, s.username, s.password);
else    xhr.open(type, s.url, s.async);
// try/catch是为防止FF3在跨域请求时报错
try {// 设定Content-Type                                          ⑦
    if (s.data)
        xhr.setRequestHeader("Content-Type", s.contentType);
        // 设定If-Modified-Since
```

```
    if (s.ifModified)
        xhr.setRequestHeader("If-Modified-Since",
            jQuery.lastModified[s.url]|| "Thu, 01 Jan 1970 00:00:00 GMT");
  // 这里是为了让服务器能判断这个请求是XMLHttpRequest
    xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
 // 设定 Accepts header 。指能接收的content-type，在服务器端设定
    xhr.setRequestHeader("Accept", s.dataType && s.accepts[s.dataType]
           ? s.accepts[s.dataType] + ", */*": s.accepts._default);
} catch (e) {}
//拦截方法，我们可以在send之前进行拦截。返回false就不send
if (s.beforeSend && s.beforeSend(xhr, s) === false) {        ⑧
    // 清除active 请求计数
    s.global && jQuery.active--;
    xhr.abort();
    return false;
}

// 触发全局的ajaxSend事件
if (s.global) jQuery.event.trigger("ajaxSend", [xhr, s]);
// 等待response返回，主要是为后面setInterval用。
var onreadystatechange = function(isTimeout) {          ⑨
// 接收成功或请求超时
if (!requestDone && xhr&& (xhr.readyState == 4 ||isTimeout == "timeout"))
    {  requestDone = true;
        //清除定时器
       if (ival) {clearInterval(ival);    ival = null;  }
    // 分析status:tiemout-->error-->notmodified-->success
    status = isTimeout == "timeout" ? "timeout" : !jQuery
        ttpSuccess(xhr) ? "error" : s.ifModified&& jQuery.
        httpNotModified(xhr, s.url)? "notmodified": "success";
    //如果success且返回了数据，那么分析这些数据
    if (status == "success") {
        try {  data = jQuery.httpData(xhr, s.dataType, s);
            } catch (e) {status = "parsererror";  }
    }
// 分析数据成功之后,进行last-modified和success的处理。
    if (status == "success") {
        var modRes;
          try {modRes = xhr.getResponseHeader("Last-Modified");
            } catch (e) { //FF中如果head取不到，会抛出异常}
         //保存last-mordified的标识。
        if (s.ifModified && modRes)jQuery.lastModified[s.url] = modRes;
       // JSONP 有自己的callback
        if (!jsonp) success();
```

```
    } else // 失败时的处理
    jQuery.handleError(s, xhr, status);
// 无论如何都进行cpmplate.timeout和接收成功
    complete();
if (s.async)  xhr = null; // 防内存泄漏
 }
};
if (s.async) {
// 这里是采用poll的方式，不是push的方式
//这里为什么不采用onreadystatechange?
var ival = setInterval(onreadystatechange, 13);
//如果过了timeout还没有请求到，会中断请求的。
    if (s.timeout > 0)
        setTimeout(function() {
            if (xhr) { xhr.abort();
                if (!requestDone) onreadystatechange("timeout");  }
        }, s.timeout);
    }
// 发送
try {xhr.send(s.data); catch(e){jQuery.handleError(s,xhr,null,e);} ⑩
// firefox 1.5 doesn't fire statechange for sync requests
if (!s.async) onreadystatechange();
function success() {
    // 调用构建请求对象时指定的success回调。
    if (s.success)    s.success(data, status);
    // 执行全局的回调
    if (s.global) jQuery.event.trigger("ajaxSuccess", [xhr, s]);
    }
function complete() {
    // 本地的回调
    if (s.complete)   s.complete(xhr, status);
    // 执行全局的回调
    if (s.global) jQuery.event.trigger("ajaxComplete", [xhr, s]);
    // 全局的ajax计数器
    if (s.global && !--jQuery.active)jQuery.event.trigger("ajaxStop");
    }
// return XMLHttpRequest便进行about()或其它操作.
return xhr;
},
```

Jquery.ajax 是大包大揽的非常复杂的一个方法。它并没有像其它的 lib 一样，把每个小部分都分开来。它是整个都整在一个函数中。看起来很多，实际上上也没有脱离前面所说的 ajax 的请求的五步。它的很大一部分代码在处理跨域请求的处理上。下面就分别就 ajax 的代码进行分析。

ajaxSettings

在①处通过继承的方式把传入参数 s 和默认的 jQuery.ajaxSettings 都 clone 到 s 变量中。S 的同名属性会覆盖 jQuery.ajaxSettings 的同名属性。这里两次继承 s,以便在测试中能检测。

```
//默认的ajax的请求参数
    ajaxSettings : {
        url : location.href,//默认是地址栏中url
        global : true,//默认支持全局的ajax事件
        type : "GET",
        timeout : 0,
        contentType : "application/x-www-form-urlencoded",
       processData : true,
        async : true,
        data : null,
        username : null,
        password : null,
        accepts : {
            xml : "application/xml, text/xml",
            html : "text/html",
            script : "text/javascript, application/javascript",
            json : "application/json, text/javascript",
            text : "text/plain",
            _default : "*/*"
        }
```

这是默认的 ajax 的设定，我们要在参数 s 设定同名的属性来覆盖这些属性。但是我们不能覆盖 accepts。这个会在后面的代码用到。我们可以通过设定 s.dataType 等于 accepts 中的某一个属性 key 指定请求的 data 类型，如 xml,html,script,json,text。dataType 还支持默认的_default 和跨域的 jsonp。不过其最终会解析成 script。

scriptTag

②~⑥是处理跨域请求的部分。对于 dataType 为 jsonp 的类型，给其请求的字符串（可能是 s.data）加上 callback=callbackfn 的 key/value 串，然后在 window 下注册一个 callbackfn 的函数。这个函数的形式如 callbackfn(data){ data = tmp;success();complete();}。它代理了通过 ajax(s) 的传入 s 参数中 success();complete() 的功能。它就是调用这个函数，实际上是调用

success();complete()的函数。

那么怎么调用呢？ajax 不支持跨域。在⑤处，我们可以看到这里是采用 scriptTag 的方式来完成。先在页面的<head>中添加一个<script src=url />的标签。因为在<head>中。浏览器会自动载入并运行请求返回的 script。如果是 jsonp 的形式，服务器端还要动态生成的 content-type 为 script 的代码：callbackfn(data);只有这样才会调用在 window 中注册的函数 callbackfn。同时传入所需要的参数。

如 dataType == "script"形式的跨域，那只能是通过 script.onload 或 script.onreadystatechange 事件来触发回调。这里我们可以通过服务器返回的 script 代码：var data=xxx。来传递参数给 s.success();s.complete()。Jquery 这里采用是全局变量 data 来进行操作的。

Ajax Event

④是采用了 jQuery.event.trigger("ajaxStart");来触发全局的 ajaxStart 事件。这也是说只要注册了这个事件的元素，在任何的 ajax 的请求时 ajaxStart 都会执行元素注册的事件处理函数。这和 Ext 的事件有点相似。但是它不是全局的。

```
jQuery.each("ajaxStart,ajaxStop,ajaxComplete,ajaxError,ajaxSuccess,ajaxSend".split(","), function(i, o) {
    jQuery.fn[o] = function(f) {// f:function
        return this.bind(o, f);
    };
```

上面的代码是为 jquery 对象注册了 ajaxStart,ajaxStop,ajaxComplete,ajaxError,ajaxSuccess,ajaxSend 这几种 ajax 的事件方法，在 jquery.ajax 中不同的时刻都会触发这些事件。当然我们也可以采用 s.global=false 来设定不触发这些事件。

因为这是全局的，个人认为其设计的目的就是为了在这些时候能以某种形式来告诉用户 ajax 的进行的状态。如在 ajaxstart 的时候，我们可能通过一个 topest 的 div 层（加上遮罩的效果）的元素注册一个 ajaxstart 事件的处理方法。该方法就是显示这个层和显示“你的数据正在提交。。。”这个的提示。这是这 6 种事件的最佳用法了。

如果进行私有处理，那么要在事件的处理函数中进行判断。因为每个事件处理函数的第二参数是 jquery.ajax(s)的 s 参数。我们可以在这个参数中做私有的标识，如 eventType:xxx。每类不同的请求有不同的 eventType 值。在事件处理函数再根据这个 eventType==xxx 进行判断，从而进行私有的处理。如果有大量的这样的私有处理也是会影响 ajax 的效率的。

setRequestHeader

⑥处是创建一个 xhr 对象并通过 open 来创建一个连接（socket）。

⑦处是设定请求的头部(setRequestHeader)。如果 data 的存在的话，那就得设定 Content-Type，便于服务器按一定的规则来解码。可以看出 post 的方式通过 data 传递数据要安全一点。

那么服务器如果区别这个请求是 ajax 呢？因为同步和异步 ajax 的请求的头文件是一样的。我们如果通过 X-Requested-With"="XMLHttpRequest”来标识这个请求是 ajax 的请求。如果服务器硬是要区分的话，就可以通过获取该头部来判断。

在头部的定义中，还可能通过 Accept 来指定接受的数据的类型，如 application/xml, text/xml", "text/html", "text/javascript, 等等。

头部还有一个 If-Modified-Since 的属性用来提高效率的。它和”Last-Modified 配合起来使用。在浏览器第一次请求某一个 URL 时，服务器端的返回状态会是 200，内容是你请求的资源，同时有一个 Last-Modified 的属性标记此文件在服务期端最后被修改的时间，格式类似这样：Last-Modified: Fri, 12 May 2006 18:53:33 GMT

客户端第二次请求此 URL 时，根据 HTTP 协议的规定，浏览器会向服务器传送 If-Modified-Since 报头，询问该时间之后文件是否有被修改过：If-Modified-Since: Fri, 12 May 2006 18:53:33 GMT 如果服务器端的资源没有变化，则自动返回 HTTP 304 （Not Changed.）状态码，内容为空，这样就节省了传输数据量。

当服务器端代码发生改变或者重启服务器时，则重新发出资源，返回和第一次请求时类似。从而保证不向客户端重复发出资源，也保证当服务器有变化时，客户端能够得到最新的资源。

拦截处理

⑧处是一个 send 之前的拦截处理，可以通过 s. beforeSend(xhr, s)函数的形式传入拦截函数。保证在发送之前确保满足某些条件。在取得返回数据的时候，也可以通过 s.dataFilter(data, type);形式来拦截处理 data。不过这里主要的作用对 data 进一步的筛选。

onreadystatechange

⑨处是 onreadystatechange 的回调处理。这里采用是 poll 的形式进行处理。它把返回的状态分成 status:tiemout-->error-->notmodified-->success—>parsererror 这几种。如果 status == "success"那么分析这些数据之后再进行 last-modified 相关的处理。为了不取回没有修改过数据。

分析数据的代码如下：

```
//处理请求返回的数据
    httpData : function(xhr, type, s) {
        var ct = xhr.getResponseHeader("content-type"),
          xml = type == "xml" || !type && ct && ct.indexOf("xml") >= 0,
           data = xml? xhr.responseXML  : xhr.responseText;
        if (xml && data.documentElement.tagName == "parsererror")
            throw "parsererror";
        //允许一个pre-filtering函数清洁repsonse
        if (s && s.dataFilter)
            data = s.dataFilter(data, type);
                //script时，就运行
        if (type == "script")    jQuery.globalEval(data);
        //json，生成json对象。
        if (type == "json")      data = eval("(" + data + ")");
        return data;

    },
```

如果返回的 content-type 是 xml,html,text 等都返回。对 script 执行 jQuery.globalEval 来执行它。对于 Json 类型，通过 eval 来生成返回的 json 对象。

```
// 在全局的范围eval 代码，也就是在<head></head>中
globalEval : function(data) {
    data = jQuery.trim(data);
        if (data) {
// Inspired by code by Andrea Giammarchi
// http://webreflection.blogspot.com/2007/08/
//global-scope-evaluation-and-dom.html
    var head = document.getElementsByTagName("head")[0]
                    || document.documentElement,
        script = document.createElement("script");
    script.type = "text/javascript";
    if (jQuery.browser.msie) script.text = data;
    else   script.appendChild(document.createTextNode(data));

    // Use insertBefore instead of appendChild to circumvent an IE6
    // bug. This arises when a base node is used (#2709).
```

```
        head.insertBefore(script, head.firstChild);
        head.removeChild(script);
            }       },
```

①②③④⑤⑥⑦⑧⑨⑩

8 FX 分析

8.1 FX 的常用方法

和前面分析的代码相比，FX 是非常让人兴奋的。以前 javaeye 登陆的时候，对其登陆窗口的淡出淡入的特效总是想入非非。Jquery 的 core 包中也提供了 Fx 的实现。

Fx 的实质是连续有序改变 dom 元素的属性达到视觉上的效果，动态地变化起来。这些属性主要是高度，宽度，透明度和颜色（背景色和前景色）等。连续有序是和时间相关的，也就是先在某个时间点改变一下 CSS 的样式属性，下一个时间点再改变一下样式属性，达到渐变的过程的效果。

Jquery 为我们提供了几种常用的 FX 的函数。

Slide 是滑出的动作，对于 slide，jquery 提供了 slideDown、slideUp、slideToggle 三种方法。slideDown 是元素向下面渐渐地滑出，最后全部可见。slideUp 是元素向上面渐渐地钻进去，最后不见。slideToggle 则是这两者之间的转换了。

Fade 是淡变的动作。对于 Fade，jquery 提供了 fadeIn、fadeOut 两个方法。fadeIn 是从无到有渐渐地显示出整个元素。fadeOut 相反，它从有到无渐渐地消失这个元素，fade 还提供了一个 fadeTo 用来改变透明性渐变到一个指定的值，而不是消失。

Jquery 还提供一组更强大的方法。Show、hide、toggle 采用一种更为优美的方式显示元素。Show 是元素的宽度渐渐变成，同时高度也渐渐变大，同时透明度也渐渐从无到不透明。这种的效果是元素从一个点渐渐变大，最终完全地显示出来。Hide 刚是相反，它是渐渐透明同时宽高逐渐变小，最后消失。Toggle 是两者之间的转换。

```
jQuery.each({
    slideDown: { height:"show" },
    slideUp: { height: "hide" },
    slideToggle: { height: "toggle" },
```

```
    fadeIn: { opacity: "show" },
    fadeOut: { opacity: "hide" }
}, function( name, props ){
    jQuery.fn[ name ] = function( speed, callback ){
        return this.animate( props, speed, callback );
    };
  });
    // 把所有匹配元素的不透明度以渐进方式调整到指定的不透明度，
     //并在动画完成后可选地触发一个回调函数。这个动画只调整元素的不透明度。
    fadeTo: function(speed,to,callback){
        return this.animate({opacity: to}, speed, callback);
    },
```

Slide 和 fade 是通过分别改变 height 或 opacity 来完成效果的。其完成的工作任务在 this.animate(props, speed, callback)上。

show

对于 show 和 hide 的，它们的效果更好一点，其代码也复杂一点。

```
  // show(speed,[callback])
    // 以优雅的动画隐藏所有匹配的元素，并在显示完成后可选地触发一个回调函数。
    // 可以根据指定的速度动态地改变每个匹配元素的高度、宽度和不透明度。
    // 显示隐藏的匹配元素 show()
show: function(speed,callback){
  return speed ?
    this.animate({height: "show", width: "show", opacity: "show"
                  }, speed, callback)
  : this.filter(":hidden").each(function(){
      this.style.display = this.oldblock || "";
        if ( jQuery.css(this,"display") == "none" ) {
          var elem = jQuery("<" + this.tagName + " />").appendTo("body");
          this.style.display = elem.css("display");// 默认的显示的display
           if (this.style.display == "none")// 显式地设定该tag不显示
                  this.style.display = "block";
              elem.remove();// 上面这些的处理有没有必要呢？
              }
           }).end();// 回到前一个jQuery对象
    },
hide: function(speed,callback){
     return speed ?
        this.animate({height: "hide", width: "hide", opacity: "hide"
              }, speed, callback)
      : this.filter(":visible").each(function(){
```

```
            this.oldblock = this.oldblock || jQuery.css(this,"display");
                this.style.display = "none";
            }).end();
    },
toggle: function( fn, fn2 ){
  return jQuery.isFunction(fn) && jQuery.isFunction(fn2) ?
    this._toggle.apply( this, arguments ) :// 原来的toggle
        (fn ?   this.animate({height: "toggle", width: "toggle",
              opacity: "toggle"}, fn, fn2)
     :  this.each(function(){jQuery(this)[ jQuery(this).is(":hidden") ?
            "show" : "hide" ]();}
            // 对每个元素都调用show,或hide函数。
            )
        );
    },
```

show 和 hide 的函数如果没有指定 speed 的参数，它们就直接地 show 或 hide 元素。没有动画的效果。如果给定的 speed 的参数。它能根据这个 speed 的值动态去改变高度宽度透明度来达到动画的 show 或 hide 的效果。Toggle 则是这两样的变换。

8.2 Fx 的核心源码分析

上面的几个常用方式都是调用了 this.animate。jquery 对象的 animate 是大包大揽的函数。所有的 FX 的效果都可以从这里得到。因为 FX 特效就是通过连续（间隔很小的时间点）去改变元素的 CSS 的样式。达到视觉上的效果。Animate 就是根据指定的参数（如 speed,元素的属性的变化范围）来完成这样的功能。

Animate

```
/**
     * 用于创建自定义动画的函数。
     * 这个函数的关键在于指定动画形式及结果样式属性对象。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”）。
     * 注意：所有指定的属性必须用骆驼形式，比如用marginLeft代替margin-left.
     * 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”或“toggle”这样的字符串值，则会为该属性调用默认的动画形式。
     * 在 jQuery 1.2 中，你可以使用 em 和 % 单位。另外，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。
     *
```

```
     * params (Options) : 一组包含作为动画属性和终值的样式属性和及其值的集合 。
duration (String,Number)
     * :(可选) 三种预定速度之一的字符串("slow", "normal", or "fast")或表示动
画时长的毫秒数值(如：1000)
     * easing (String) : (可选) 要使用的擦除效果的名称(需要插件支持).默认jQuery
提供"linear" 和 "swing".
     * callback (Function) : (可选) 在动画完成时执行的函数
     */
animate: function( prop, speed, easing, callback ) {
    var optall = jQuery.speed(speed, easing, callback); ①
   // 执行each或queue方法
return this[ optall.queue === false ? "each" : "queue" ](function(){②
    var opt = jQuery.extend({}, optall), p,
    hidden=this.nodeType==1&&jQuery(this).is(":hidden"),//元素节点且隐藏
    self = this;// 当前的元素
    for ( p in prop ) {                                          ③
      //如果是完成的状态，就直接调用complate函数
      if ( prop[p] == "hide" && hidden || prop[p] == "show" && !hidden )
          return opt.complete.call(this);
     if (( p == "height" || p == "width")&& this.style ){//style高宽度
        opt.display = jQuery.css(this, "display");// 保存当前元素的display
        opt.overflow = this.style.overflow;// 保证没有暗中进行的
        }
     }
    if ( opt.overflow != null )// 超出部分不见
        this.style.overflow = "hidden";
    opt.curAnim = jQuery.extend({}, prop);//clone传入的参数prop
    jQuery.each( prop, function(name, val){                      ④
       // 对当前元素的给定的属性进行变化的操作
       var e = new jQuery.fx( self, opt, name );                 ⑤
      // 传参的属性可以用toggle，show，hide，其它
      // 调用当前e.show,e.hide,e.val的方法，jQuery.fx.prototype
        if ( /toggle|show|hide/.test(val) )                      ⑥
           e[ val == "toggle" ? hidden ? "show" : "hide" : val ]( prop );
       else {                                                    ⑦
           var parts = al.toString().match(/^([+-]=)?([\d+-.]+)(.*)$/),
           start = e.cur(true) || 0;// 当前元素当前属性的值
          // +=" 或 "-=" 来让元素做相对运动。
           if ( parts ) {                                        ⑧
               var end = parseFloat(parts[2]), unit = parts[3] || "px";
               if ( unit != "px" ) {// 计算开始的值=(end/cur)*start
                   self.style[ name ] = (end || 1) + unit;
                   start = ((end || 1) / e.cur(true)) * start;
                   self.style[ name ] = start + unit;
```

```
                }
            if( parts[1]) end = ((parts[1] == "-="? -1 : 1)* end)+ start;
            e.custom( start, end, unit );// 动画
         }
        //直接计算start和end的位置来动画。val是数值的end,start当前的属性值。
        else   e.custom( start, val, "" );// 动画    ⑨
        }
    });
    / For JS strict compliance
return true;
}); },
```

Animate 通过传入的参数对于 jquery 对象中的每个元素的每个指示的属性都进行时间上渐变的改变。下面就分析其中的代码。

jQuery.speed

在①是就通过 jquery.speed 来进行参数的统一整理。

```
// 主要用于辅助性的工作
speed: function(speed, easing, fn) {
    var opt = speed && speed.constructor == Object ? speed : {
   // coplete是至多三个参数的最后一个。看看是否传入动画完成回调的函数
    complete: fn ||!fn && easing ||jQuery.isFunction( speed ) && speed,
    duration: speed,// 持继的时间，动画运行的时间。
   //找到动画中属性随时间渐变的的算法。
   easing:fn&&easing || easing && easing.constructor != Function && easing
    };
//计算出正确的duration，它支持fast,slow这样已经定义的常量
 opt.duration = (opt.duration &&
     (opt.duration.constructor == Number?
         opt.duration :
        jQuery.fx.speeds[opt.duration]))||jQuery.fx.speeds._default;
       // Queueing
       opt.old = opt.complete;
//这里是把complete形成了包裹，看看参数中是否指定队列操作，
//如果先出列，再执行complete
opt.complete = function(){
   //可能通过参数指定queue，this指向是当前的dom元素。
    if ( opt.queue !== false )  jQuery(this).dequeue();//出queue
    if (jQuery.isFunction( opt.old ))  opt.old.call( this );
    };

return opt;
```

```
},
```

jquery.speed 是对参数进行管理的操作函数。它首先看看 speed 是不是紧缩型的参数如{ complete:xx, easing:xx, duration:xx}。如果不是，就根据传入的参数进行判断，组成紧缩型的对象参数。

由于 jquery 对象支持 fast,slow 这样的常量来代替具体的 speed 的数值，第二步就是进行 speed 的处理。如果没有提供就提供默认的 speed。其代码在 jQuery.fx.speeds 中 speeds:{ slow: 600, fast: 200, _default: 400},定义了三种形式的速度。

接下来是对在完成的时间执行的 complete 的回调函数进行包裹。包裹就是看看参数中提供了队列的操作没有。提供了就进行出队列的操作。之后再执行 complete 的回调函数。

```
jQuery.fn.dequeue = function(type){
    type = type || "fx";
    return this.each(function(){
        var q = queue(this, type);// 得取type的的值
        q.shift();// 移出一个
        if ( q.length )
            q[0].call( this );
    });
};
```

jQuery.fn.dequeue 根据 type 取到当前 dom 元素的 queue。先移出一个。再运行 queue 中的第一个。

Queue

在②处和上一部分都看看参数中提供了队列的操作没有，有就是进行 queue 的操作。Queue 的操作和 each 的操作是不一样的。

```
// 实现队列操作，为jQuery对象中的每个元素都加type的属性，值为fn.
 queue: function(type, fn){
    // 可能看出支持一个参数的fn或array形式的集合其type为默认的fx的形式。
    if(jQuery.isFunction(type)||( type && type.constructor == Array)) {
            fn = type; type = "fx";  }
    //没有参数时或一个参数是字符的type时就从jquery对象第一个元素的data中取
    //出相对于的type(空就是所有)的队列中的元素（fn）。
    if ( !type || (typeof type == "string" && !fn) )
        return queue( this[0], type );//从queue取出
    //把fn保存在jquery对象中的每个元素的data中的对应的type中去。
    //对于fn是fn的数组集合，就直接设定data中type为fn的数组
    //如果是单个的fn,那么就采用追加的形式追加到data的type的fn数组中。
    //如果追加的形式，而且之前数组中没有fn元素。那么就执行这个元素。
```

```
return this.each(function(){
 if ( fn.constructor == Array )// 数组的形式
      queue(this, type, fn);// 存储在元素的type属性中
 else {
       queue(this, type).push( fn );
         if ( queue(this, type).length == 1 )  fn.call(this);
         }
     });
 },
```

分析上面的代码可以看出 Queue 的操作和 each 的操作是不太一样的。Each 是对每个元素都执行 fn 函数。而 queue 对于每个元素都把 fn 函数放到自已对应的 cache 中 fx 的属性中保存，是单个的 fn 的话，也立马运行。在②处和 each 的作用差不多。

在上面的代码，它还调用了 queue(this, type, fn);来实现把把 fn 存到 this 元素的 data 中的对应的 type 中去。

```
// 为元素加上type的array的属性，或返回取到elem上type的值
var queue = function( elem, type, array ) {
    if ( elem ){
        type = type || "fx";
        var q = jQuery.data( elem, type + "queue" );
        if ( !q || array )
        q = jQuery.data( elem, type + "queue", jQuery.makeArray(array) );
    }
    return q;
  };
```

这个全局的函数就是在 jQuery.data 进一步封装，使它支持默认的 fx type 和 array 的类数组的参数。

Jquery 这里的 Queue 实质上没有起到什么作用。它本来可以支持一个元素的的连续执行几个 Queue 的函数达到更丰富的动画的效果。估计这里是没有完成。

①②③④⑤⑥⑦⑧⑨⑩

jQuery.fx

⑤处 animate 为 dom 元素的每个指定的属性都生成了一个 fx 对象。

```
// 根据参数构成一个对象
 fx: function( elem, options, prop ){
     this.options = options;
     this.elem = elem;
     this.prop = prop;
```

```
        if ( !options.orig )
            options.orig = {};
    }
```

Fx 函数是一个构建函数，仅仅是把参数变成本对象的属性。以便于 fx 对象的其它方法来使用这些参数。在⑥处就是通过调用 fx 的方法 show 或 hidden 来完成一个属性的逐渐的改变。在⑨是通过 custom 方法来直接进行一个动画。

先看一下show或hidden：

```
show: function(){
 // 保存当前的，以被修改之后能得到初始的值
 this.options.orig[this.prop]=jQuery.attr(this.elem.style,this.prop);
 this.options.show = true;//标明是进行show操作
 this.custom(0, this.cur());
 //让最开始时以1px的宽或高度来显示。防止内容flash
 if ( this.prop == "width" || this.prop == "height" )
        this.elem.style[this.prop] = "1px";
 jQuery(this.elem).show();
    },
hide: function(){
 // 保存当前的，以被修改之后能得到初始的值
 this.options.orig[this.prop]=jQuery.attr(this.elem.style,this.prop);
 this.options.hide = true;//标识是进行hide操作
 this.custom(this.cur(), 0);},
```

show 和 hide 是在指定元素的属性为 show 或 hide 的时候调用的，如 height: "show", width: "show", opacity: "show"。它们都是先保存原始的改悔。之后调用 custom 来完成动画。和⑨处是一样的。

也就是说完成动画的工作在 custom 中：

```
   // 开动一个动画
    custom: function(from, to, unit){
        this.startTime = now();//动画开始的时候
        this.start = from;//位置开始点
        this.end = to;//位置结果点
        this.unit = unit || this.unit || "px";
        this.now = this.start;//位置当前点
        //state是时间间隔在总的duration的比率
        //pos是按一定算法把时间上的比率折算到位置上的比率
        this.pos = this.state = 0;
        //根据this.now位置当前点的值来设定元素的属性显示出来
        this.update();

        var self = this;
        function t(gotoEnd){
            return self.step(gotoEnd);// 调用step(gotoEnd)//本对象的
```

```
        }
        t.elem = this.elem;//删除的时候做判断用
        //timers栈是公共的，不同的元素的不同的属性step都是放在这里面。
        jQuery.timers.push(t);

        if ( jQuery.timerId == null ) {
            jQuery.timerId = setInterval(function(){
                var timers = jQuery.timers;
            //倒是觉得这里会有同步冲突的问题。Ext.observable中就有解决方法
                for ( var i = 0; i < timers.length; i++ )
                //当一个属性的动画完成，或强迫完成的时候，把step从数组中删除.
                //同时把i的位置不改变。继续下一个。
                if ( !timers[i]() )  timers.splice(i--, 1);
                  //说明还有属性的动画没有完成，step还在timers中。
                  //那么就不clearInterval，13ms之后再继续。直到数组
                  //中所有的step都被删除。
                if ( !timers.length ) {
                    clearInterval( jQuery.timerId );
                    jQuery.timerId = null;
                }
            }, 13);
        }

    },
```

在 custom 中为 fx 对象动态地追加了几个属性。Start 和 end 属性指的属性值变化的开始和结束位置。这是属性值发生变化的范围。Now 是在 Start 和 end 范围的某一个点，也就是当前要比属性设定值。这个值是根据时间的间隔比率再通过一定的算法来得出的。pos 和 state 一个是位置上的比率，一个是时间上比率，值在 0~1 之间。

Custom 通过 this.now = this.start;和 this.update();来设起始位置的样式属性。

```
    // 为元素设值，更新显示
    update: function(){
        //可以在显示之前进行自定义的显示操作
        //这里可以是改变this.now或元素的其它属性。
        //改变this.now是改变动画的轨迹，改变其它的属性会有更多的效果
        if ( this.options.step )
            this.options.step.call( this.elem, this.now, this );
        //根据this.now来改变/设值当前属性的值。也就改变了样式。
        (jQuery.fx.step[this.prop] || jQuery.fx.step._default)( this );

        // 对于高度和宽度，肯定是要能看出效果的，故采用display=block。
        if ( ( this.prop == "height" || this.prop == "width" )
                        && this.elem.style )
```

```
            this.elem.style.display = "block";
    },
```

通过 udate 设定起始位置的样式属性之后，custom 第二步就是采用 setInterval 每隔 13ms 就执行一次当前的 fx 对象的 step 方法。该方法实现了动画结束的扫尾工作和动画过程中按一定的算法来计算 this.now 的值。因为这个值的改变，之后调用 update 就可以改变样式的属性。

这样一来，就实现了元素的某个属性的渐变过程。

```
// 动画的每一个步骤
    step: function(gotoEnd){
        var t = now();//运行到当前的时间，因为是13ms才运行一次。
        // 强行指定结束或当前时间大于startTime+duration
        if ( gotoEnd || t > this.options.duration + this.startTime ) {
            this.now = this.end;//当前的位置为结束位置
            this.pos = this.state = 1;//当前的state,pos的比率为1.最大。
            this.update();//显示
           //标识这个属性的动画已经完成
            this.options.curAnim[ this.prop ] = true;
           //再一次判断是否完成
            var done = true;
            for ( var i in this.options.curAnim )
                if ( this.options.curAnim[i] !== true )
                    done = false;

            if ( done ) {
                if ( this.options.display != null ) {//  恢复overflow
                    this.elem.style.overflow = this.options.overflow;
                    // 恢复 display
                    this.elem.style.display = this.options.display;
                    //判断其是否恢复成功，
                    if ( jQuery.css(this.elem, "display") == "none" )
                        this.elem.style.display = "block";
                }

                //如果是hide的操作
                if ( this.options.hide )
                    this.elem.style.display = "none";

                    //如果元素已经show或hide,恢复其动画改变的属性
                if ( this.options.hide || this.options.show )
                    for ( var p in this.options.curAnim )
                        jQuery.attr(this.elem.style, p,
                                    this.options.orig[p]);
            }
```

```
            if ( done )// 运行complete的回调函数
                this.options.complete.call( this.elem );

            return false;
        } else {
            var n = t - this.startTime;//时间间隔
            this.state = n / this.options.duration;//时间间隔比率

            //根据时间间隔的比率再按一定的算法比率来计算
            //当前的运动的位置点的比率。默认是swing的算法
            this.pos = jQuery.easing[this.options.easing ||
            (jQuery.easing.swing ? "swing" : "linear")]
                    (this.state, n, 0, 1, this.options.duration);
            //当前的位置
            this.now = this.start + ((this.end - this.start) * this.pos);

            // 显示
            this.update();
        }

        return true;
    }
```

Step 中第一部分是完成时的扫尾工作，恢复动画时改变的属性和运行 complete 的回调函数。第二部分就是计算 this.now 的值之后显示出来。目前 jquery 提供了两种算法来计算从时间间隔比率转换成位置上的比率。

```
    easing: {
        linear: function( p, n, firstNum, diff ) {
            return firstNum + diff * p;
        },
        swing: function( p, n, firstNum, diff ) {
            return ((-Math.cos(p*Math.PI)/2) + 0.5) * diff + firstNum;
        }

    },
```

这就是它的算法。对于 linear 的形式，Jquery 采用了 1:1 的关系来计算的。

Stop

一个动画，有的时间可能会出现在没有运行完成就中断。Stop 就是对这个进行操作的。

```
stop: function(clearQueue, gotoEnd){
```

```
var timers = jQuery.timers;
if (clearQueue)  this.queue([]);// 清除
    this.each(function(){
    //倒序是为了能把在loop过程加入timers的当前元素的属性的动画step也给删除。
        for ( var i = timers.length - 1; i >= 0; i-- )
            if ( timers[i].elem == this ) {
                if (gotoEnd) timers[i](true); // 强迫动画结束
                 timers.splice(i, 1);
            }
    });
    // start the next in the queue if the last step wasn't forced
    if (!gotoEnd)    this.dequeue();

    return this;
}
```

在 stop 中，我们可以看出 step（force）中的参数的作用。这个强迫动画到最后一步，即动画结束进行扫尾工作。这里的 jQuery.timers 是公共的集合。在 custom 中的 setInterval 就是对它进行操作的。