几乎很难从 jQuery 分离其中的一部分功能。所以在这里我分享下应该读 jQuery 源码的一些成果，以及读源码的方法。啃代码是必须的。

1. 代码折叠是必须的。

因此必须在支持语法折叠的编辑器里打开源码。 根据折叠层次，我们可以很快知道：所有 jQuery 的代码都在一个函数中：

```
(function( window, undefined ) {
// jQuery 代码


})(window);
```

这样可以避免内部对象污染全局。传入的参数 1 是 window， 参数 2 是 undefined ， 加快 js 搜索此二对象的速度。

2. 接着打开第一级折叠。

可以发现 jQuery 代码是按这样顺序来组织：

- 定义 jQuery 函数 (代码 20 - 1081 行)
- 生成 jQuery.support (代码 1083 - 1276 行)
- 和 data 有关扩展 (代码 1279 - 1510 行)
- 和队列有关扩展 (代码 1514 - 1605 行)
- 和属性有关扩展 (代码 1609 - 1988 行)
- 和事件有关扩展 (代码 1993 - 3175 行)
- 内部的 Sizzle CSS Selector Engine (代码 3183 - 4518 行)
- 和节点有关扩展 (代码 4520 - 5492 行)
- 和样式有关扩展 (代码 5497 - 5825 行)
- 和 ajax 有关扩展 (代码 5830 - 7172 行)
- 和效果有关扩展 (代码 7176 - 7696 行)
- 和定位有关扩展 (代码 7700 - 8065 行)

下面的模块可以用上面的模块，上面的模块不需要下面的模块

3. 定义 jQuery 函数 (代码 20 - 1081 行)

总的代码是这样的框架：

```
var jQuery = (function() {
 // 创建 jQuery 对象
var jQuery = function( selector, context ) {
 // 略


 };


 // 创建 jQuery.fn 对象
 jQuery.fn = jQuery.prototype = {
 // 略
 };
```

```
// 声明 jQuery.extend
jQuery.extend = jQuery.fn.extend = function() {
// 略
};

// 使用 jQuery.extend 扩展自己
jQuery.extend({
// 略

});

// 浏览器方面的一些琐碎
// 略

// 定义全局对象
return (window.jQuery = window.$ = jQuery);

})();
```

从这里知道: 平时所用的 $ 其实就是 jQuery 函数的别名。

jQuery 对象似乎一直都是这东西:

```
var jQuery = function( selector, context ) {
// 实际上 jQuery 对象是 jQuery.fn.init 返回的。
return new jQuery.fn.init( selector, context, rootjQuery );
}
```

这个函数表示: 要想知道函数 jQuery 是什么东西，必须看 jQuery.fn.init 对象。

同时这也解释了为什么写代码不需要 new jQuery。

再看第 29 行 - 97 行， 都是一些变量声明，这些变量在下面的函数用到。提取变量的好处: 对正则节约编译的时间， 同时能在压缩的时候获得更小的结果。

这个 fn 其实是 jQuery.prototype ，这也是为啥 jQuery.fn 就是扩展 jQuery 对象的唯一原因。

肯能有人会疑问， jQuery 返回 new jQuery.fn.init， 也就是说，平时的函数应该是 jQuery.fn.init.prototype 所有的成员，不是 jQuery.prototype 成员。当然原因也很简单: jQuery.fn.init.prototype === jQuery.prototype (代码 322 行)

jQuery 对 js 对象处理和中国人讲话一样绕。这里总结下到底 jQuery 对象是个什么家伙。

jQuery 是普通函数， 返回 jQuery.fn.init 对象的实例(new jQuery.fn.init())。

然后 jQuery.fn === jQuery.prototype === jQuery.fn.init.prototype ，最后， jQuery 返回的对象的成员和 jQuery.fn 的成员匹配。

jQuery.fn 下有很多成员，下面稍作介绍:

init - 初始化(下详细说明)

constructor - 手动指定一个构造函数。 因为默认是 jQuery.fn.init

length - 让这个对象更接近一个 原生的数组

size - 返回 length

toArray - 通过 Array.prototype slice 实现生成数组

get - 即 this[num] ，当然作了下 参数索引 的处理。

pushStack - 加入一个元素

ready - 浏览器加载后执行(下详细说明)

end - 通过保存的 prevObject 重新返回

each - 参考 http://www.cnblogs.com/Fooo/archive/2011/01/11/1932900.html

参考 http://www.cnblogs.com/rubylouvre/archive/2009/11/21/1607632.html

jQuery.fn.init 就是所谓的 $ 函数。 也就是说，平常的 $("#id") 就是 new jQuery.fn.init("#id");

这个函数很长，但代码覆盖率小。

```
init: function( selector, context, rootjQuery ) {
// 参数: selector 选择器
// context 上下文
// rootjQuery 父节点

// 处理 $(""), $(null) 和 $(undefined)
if ( !selector ) {
return this;
}

// 处理 $(DOMElement)
if ( selector.nodeType ) {

// 直接扔数组中， 就搞定了。
this.context = this[0] = selector;
this.length = 1;
return this;
}
```

```
// 处理 $("body") body 元素只存在一次，单独找它
if ( selector === "body" && !context && document.body ) {

// 同样扔数组中， 顺便把 selector 更新更新。
this.context = document;
this[0] = document.body;
thisis.selector = "body";
this.length = 1;
return this;
}

// 处理 $(HTML 代码 或者是 css 选择器)
if ( typeof selector === "string" ) {
// 略

// 处理 $(函数)
} else if ( jQuery.isFunction( selector ) ) {

// 如果是函数，则执行 $(document).ready , 这样 $(document).ready(func) 简为 $(func)
return rootjQuery.ready( selector );
}

// 略。

// 如果传入的是一个 Dom 列表 ( getElementsByTagName 结果 ) 则转为 jQuery 数组。
return jQuery.makeArray( selector, this );
}
// 这部分代码是 上段中 略 的 也就是说是 jQuery(字符串) 处理。

// 检查是否字符串是常用选择器
(/^(?:[^<]*(<[\w\W]+>)[^>]*$|#([\w\-]+)$)/)
match = quickExpr.exec( selector );

// 检查是否正确匹配
```

```
if ( match && (match[1] || !context) ) {

// 处理: $(html) -> $(array)
if ( match[1] ) {

// 获取正文，默认 document
context = context instanceof jQuery ? context[0] : context;
doc = (context ? context.ownerDocument || context : document);

// 如果传入简单的 "<标签>"， ( /^<(\w+)\s*\/?>(?:<\/\1>)?$/)
ret = rsingleTag.exec( selector );

// 返回 createElement("tag")

return jQuery.merge( this, selector );

// 处理: $("#id")
} else {
elem = document.getElementById( match[2] );

// 因为有的浏览器 getElementById 不只返回 id 匹配的，所以做检查。
return this;
}

// 处理 $("标签")
} else if ( !context && !rnonword.test( selector ) ) {
this.selector = selector;
this.context = document;
selector = document.getElementsByTagName( selector );
return jQuery.merge( this, selector );

//处理: $(选择器, $(...))
} else if ( !context || context.jquery ) {
return (context || rootjQuery).find( selector );

// 处理: $(选择器, 上下文)
```

```
// (相当于: $(上下文).find(选择器)
} else {
return this.constructor( context ).find( selector );
}
```

这个函数用于 扩展函数
函数中含多个参数判断，为了使用可以更灵活。
基本原理就是 for(in)，这里不具体介绍了。

```
noConflict: function( deep ) {
 window.$ = _$;

 if ( deep ) {
 window.jQuery = _jQuery;
 }

 return jQuery;
 },
```

不多解释了，就是让 jQuery 恢复为全局的对象。
其中有 2 个函数:
jQuery.ready 触发执行 readyList 中的所有函数
jQuery.bindReady 初始化让 jQuery.ready 成功执行

```
bindReady: function() {

 // 如果已经执行 bindReady 则返回。
 if ( readyBound ) {
 return;
 }

 readyBound = true;

 // 如果页面已经加载， 马上执行 jQuery.ready
 if ( document.readyState === "complete" ) {
 return setTimeout( jQuery.ready, 1 );
 }
```

```
// 标准浏览器支持 DOMContentLoaded
if ( document.addEventListener ) {
// 你懂的
document.addEventListener( "DOMContentLoaded", DOMContentLoaded,
false );

// 为什么还要 load ？， 因为有些时候 DOMContentLoaded 失败(如 iframe) ，
而 load 总是会成功，所以,同时处理 DOMContentLoaded load，在 jQuery.ready
中会删除监听函数，保证最后这个函数只执行一次
window.addEventListener( "load", jQuery.ready, false );

// IE 浏览器( IE 8 以下)
} else if ( document.attachEvent ) {

// 使用 onreadystatechange
document.attachEvent("onreadystatechange", DOMContentLoaded);

// 同理
window.attachEvent( "onload", jQuery.ready );

// 如果 IE 下且非 iframe，这里有个技巧。 见 doScrollCheck();
// 原理：浏览器在没加载时 设置 scrollLeft 会错误，所哟每隔 1 秒厕所 是否
scrollLeft 成功，如果发现成功，则执行 jQuery.ready 。但这只对非 frame 会有
用。
var toplevel = false;

try {
toplevel = window.frameElement == null;
} catch(e) {}

if ( document.documentElement.doScroll && toplevel ) {
doScrollCheck();
}
}
},
```

bindReady 函数在执行 ready 时执行。(jQuery 1.4 之前版本都是 绝对执行，不管需不需要 ready 函数)

4. 生成 jQuery.support (代码 1083 - 1276 行)

人人都说 jQuery.support 是个好东西。确实，这东西可以解决很多兼容问题。

jQuery.support 是基于检测的浏览器兼容方式。也就是说，创建一个元素， 看这个元素是否符合一些要求。

比如测试元素是否支持 checkOn 属性，只要先 set check = 'on' 然后看浏览器是否 get check == 'on'。

此部分源码不具体介绍了。

5.和 data 有关扩展 (代码 1279 - 1510 行)

jQuery 的 data() 用于存储一个字典。而这些数据最后都保存在 jQuery.cache (代码 1283 行) , 全局对象存在 windowData (代码 1279 行) 。

但如果正确根据对象找到其在 jQuery.cache 的存储对象? 这就是 expando 字符串。

比如一个对象: elem 。

满足: elem.expando = "jQuery12321";

那么 jQuery.cache["jQuery12321"] 就是存储这个 elem 数据的对象。

实际上， 不是 elem.expando 表示键值，而是 elem[jQuery.expando] 表示。

而一个对象数据又是一个字典，所以最后执行 jQuery.data(elem, 'events') 后就是:

jQuery.cache[elem[jQuery.expando]]['events'] 的内

容 (jQuery.cache[elem[jQuery.expando]] = {})

6.和队列有关扩展 (代码 1514 - 1605 行)

队列是 jQuery 1.5 新增的。

主要用于特效等需要等待执行的时候。

队列主要操作就是 进队 queue 出队 dequeue

jQuery 队列内的数据:

如果没有执行:

[将执行的 1， 将执行的 2]

现在开始执行 <将执行的 1>， 如果 type 为空或 "fx", 队列内数据:

["inprogress", 将执行的 2]

执行完之后:

["将执行的 2]

以上的这些数据都存在 jQuery.data(obj, (type || "fx") + "queue") (代码 1520 - 1522 行)

jQuery.delay 则用于延时执行一个函数。相当于把原来队列更新为 setTImeout 后的结果。(代码 1589 -1598 行)

7.和属性有关扩展 (代码 1609 - 1988 行)

从这里开始，需要了解一个函数 jQuery.access。对于 attr ，css 之类的函数，如果需要返回值，只返回第一个元素的值，如果是设置值，则设置每个元素的值。这个神奇的效果就是 jQuery.access 搞定的。

jQuery.access 代码在 794 - 819 行

jQuery 大部分函数都是依赖 jQuery.access 实现的，比如有一个函数 XX，对用户而言，调用的是 jQuery.fn.XX, 而这个函数需要对多个元素(jQuery 数组内的所有的节点) 操作，或者对 1 个元素操作，通过 jQuery.access 转换(不一定都是)，最后只写对 1 个元素的操作。这 1 个元素的操作往往是 jQuery.XX 函数，因此，我们往往能看到即存在 jQuery.XX, 又存在 jQuery.fn.XX, 而其实 jQuery.fn.XX 都是依靠 jQuery.XX 的，或者说 jQuery.XX 是底层函数， jQuery.fn.XX 是方便用户的工具 。

jQuery.fn.attr 这个函数(代码 1632 - 1634 行) 只有 1 句话，真正的实现是 jQuery.attr (代码 1880 - 1988)

又是一个大于 100 行的函数

```
attr: function( elem, name, value, pass ) {
 // 检查是否为 nodeType 为 Element
 if ( !elem || elem.nodeType === 3 || elem.nodeType === 8 ||
elem.nodeType === 2 ) {
 return undefined;
 }

 // 如果这个属性需要特殊对待。 height/width/left 等属性需特殊计算
 if ( pass && name in jQuery.attrFn ) {
 return jQuery(elem)[name](value);
 }

 // 检查是否为 XML 还 HTML
 var notxml = elem.nodeType !== 1 || !jQuery.isXMLDoc( elem ),
 // Whether we are setting (or getting)
 set = value !== undefined;

 // 修正名字， 比如 float 改 cssFloat
 name = notxml && jQuery.props[ name ] || name;

 // 只有在节点的时候执行。
 if ( elem.nodeType === 1 ) {
 // 在 IE7- 下， href src 属性直接获取会返回绝对位置，而不是真实的位置字符串，
```

```
// 要获得它们的真实值，需要 elem.getAttribute("href", 2);
var special = rspecialurl.test( name );

// Safari 误报默认选项。通过获取父元素的已选择索引来修复。
if ( name === "selected" && !jQuery.support.optSelected ) {
var parent = elem.parentNode;
if ( parent ) {
parent.selectedIndex;

// 对 optgroups ，同理
if ( parent.parentNode ) {
parent.parentNode.selectedIndex;
}
}
}

// 检查属性是否存在， 有些时候 name in elem 会失败，所以多次测试。
if ( (name in elem || elem[ name ] !== undefined) && notxml && !special )
{

// 如果设置属性
if ( set ) {
// 在 IE， 不能设置属性 type 。
if ( name === "type" && rtype.test( elem.nodeName ) && elem.parentNode )
{
jQuery.error( "type property can't be changed" );
}

// 如果 value === null， 表示移除属性
if ( value === null ) {
if ( elem.nodeType === 1 ) {
elem.removeAttribute( name );
}

} else {
```

```
// 一切属性设置就是 1 句话。。。
elem[ name ] = value;
}
}

// 表单索引元素获取需要 getAttributeNode( name ).nodeValue
if ( jQuery.nodeName( elem, "form" ) && elem.getAttributeNode(name) )
{
return elem.getAttributeNode( name ).nodeValue;
}

// elem.tabIndex 特殊处理
if ( name === "tabIndex" ) {
var attributeNode = elem.getAttributeNode( "tabIndex" );

return attributeNode && attributeNode.specified ?
attributeNode.value :
rfocusable.test( elem.nodeName ) || rclickable.test( elem.nodeName )
&& elem.href ?
0 :
undefined;
}

return elem[ name ];
}

// 处理 style 属性
if ( !jQuery.support.style && notxml && name === "style" ) {
if ( set ) {
elem.style.cssText = "" + value;
}

return elem.style.cssText;
}

if ( set ) {
```

```
// 这里除了 IE， 其它属性使用标准 setAttribute
elem.setAttribute( name, "" + value );
}

// 如果属性不存在，返回 undefined， 而不是 null 或 "" 之类的。
if ( !elem.attributes[ name ] && (elem.hasAttribute
&& !elem.hasAttribute( name )) ) {
return undefined;
}

// 见上
var attr = !jQuery.support.hrefNormalized && notxml && special ?
// Some attributes require a special call on IE
elem.getAttribute( name, 2 ) :
elem.getAttribute( name );

// 同上
return attr === null ? undefined : attr;
}
// 如果不是 DOM 元素，检查处理。
if ( set ) {
elem[ name ] = value;
}
return elem[ name ];
}
```

windowData

8.和事件有关扩展 (代码 1993 - 3175 行)

平时我们都是调用 click(func) 之类的函数， 而其实这些都是工具函数，真正和事件挂钩的函数是

jQuery.fn.bind - 调用 jQuery.event.add

jQuery.fn.unbind - 调用 jQuery.event.remove

jQuery.fn.trigger - 调用 jQuery.event.trigger

jQuery.fn.one - 调用 jQuery.fn.bind，Query.fn.unbind

要想知道 jQuery 的事件原理，必须读 jQuery.event.add (代码 2012 - 2155 行)

```
add: function( elem, types, handler, data ) {

 // 只对节点操作。
 if ( elem.nodeType === 3 || elem.nodeType === 8 ) {
 return;
 }

 // IE 无法传递 window，而是复制这个对象 。
 if ( jQuery.isWindow( elem ) && ( elem !== window
&& !elem.frameElement ) ) {
 elem = window;
 }

 // 如果 handler === false， 也就是说就是阻止某事件，
 // 这样只要 bind("evt", false); 就是阻止此事件。
 if ( handler === false ) {
 handler = returnFalse;
 } else if ( !handler ) {
 return;
 }

 // handleObjIn 是内部处理句柄， handleObj 是直接使用的处理句柄。
 var handleObjIn, handleObj;

 if ( handler.handler ) {
 handleObjIn = handler;
 handler = handleObjIn.handler;
 }

 // 为函数生成唯一的 guid 。具体下面介绍。
 if ( !handler.guid ) {
 handler.guid = jQuery.guid++;
 }

 // 获取一个节点的数据。
 var elemData = jQuery.data( elem );
```

```
// 如果没有数据，则直接返回。
if ( !elemData ) {
return;
}

// 避免和原生的 js 对象混淆。
var eventKey = elem.nodeType ? "events" : "__events__",

// 这里就是关键。
// elemData 是存储数据的位置， 而 elemData[ eventKey ] 就是存储当前事件的对象。 elemData.handle 就是当前绑定的所有函数数组。
// 也就是说，当我们绑定一个函数时，会往 elemData.handle 放这个函数，然后事件触发时，会遍历 elemData.handle 中函数然后去执行。
// 肯能有人会问，为什么这么做，因为原生的 DOM 内部也有一个 函数数组，事件触发后会执行全部函数。答案还是 兼容。
// 标准浏览器使用 addEventListener
// IE 使用 attachEvent
// 而这 2 者还是有差距的。因为 addEventListener 执行函数的顺序即添加函数的顺序，然而 attachEvent 执行函数的顺序和添加的顺序是相反的。
// jQuery 使用自定义的 handler 数组，好处有：
// 因为最后仅绑定一次原生事件，事件触发后，手动执行 数组中的函数。这样保证兼容。
// 同时也可以知道到底绑定了什么函数，可以方便 trigger 函数的完成。

events = elemData[ eventKey ],
eventHandle = elemData.handle;

// 一些功能。。
if ( typeof events === "function" ) {
eventHandle = events.handle;
events = events.events;

} else if ( !events ) {
if ( !elem.nodeType ) {
elemData[ eventKey ] = elemData = function(){};
}
```

```
elemData.events = events = {};
}

// 如果是第一次执行，需创建 eventHandle
if ( !eventHandle ) {

// eventHandle 就是真正绑定到原生事件的那个函数，这个函数用来执行
events.hadlers 用。
elemData.handle = eventHandle = function() {
// Handle the second event of a trigger and when
// an event is called after a page has unloaded
return typeof jQuery !== "undefined" && !jQuery.event.triggered ?
jQuery.event.handle.apply( eventHandle.elem, arguments ) :
undefined;
};
}

// 绑定函数和原生，这样可以保证函数可执行为目前作用域。
eventHandle.elem = elem;

// 处理 jQuery(...).bind("mouseover mouseout", fn);
types = types.split(" ");

var type, i = 0, namespaces;

while ( (type = types[ i++ ]) ) {
handleObj = handleObjIn ?
jQuery.extend({}, handleObjIn) :
{ handler: handler, data: data };

// 略

// 绑定 type guid
handleObj.type = type;
if ( !handleObj.guid ) {
```

```
handleObj.guid = handler.guid;
}

// 获取当前的函数数组。
var handlers = events[ type ],
special = jQuery.event.special[ type ] || {};

// 如果第一次，则创建这个数组。
if ( !handlers ) {
handlers = events[ type ] = [];

// 特殊事件要执行 setup 而不是标准 addEventListener。
// 此行用来支持自定义的事件。
if ( !special.setup || special.setup.call( elem, data, namespaces,
eventHandle ) === false ) {
// 标准事件。 这里绑定的为 eventHandle
if ( elem.addEventListener ) {
elem.addEventListener( type, eventHandle, false );

} else if ( elem.attachEvent ) {
elem.attachEvent( "on" + type, eventHandle );
}
}
}

// 自定义事件，执行 add
if ( special.add ) {
special.add.call( elem, handleObj );

if ( !handleObj.handler.guid ) {
handleObj.handler.guid = handler.guid;
}
}

// 不管是不是首次，都放入目前绑定的函数。
handlers.push( handleObj );
```

```
// 为实现 trigger 。
jQuery.event.global[ type ] = true;
}

// 让 IE 下可以正常回收 elem 内存。
elem = null;
},
```

框架的义务当然也包括 事件参数的修复。jQuery 是自定义事件对象， 这个对象模拟真实事件对象。

根据上文可以知道，真正绑定事件的是一个函数，这个函数执行时会先 生成自定义事件对象， 然后把此对象作为参数调用所有的 handler 。

jQuery 自定义事件是 jQuery.Event 。 (代码 2583-2610 行)

```
jQuery.Event = function( src ) {
// 支持 没有 new jQuery.Event
if ( !this.preventDefault ) {
return new jQuery.Event( src );
}

// 略

}
```

这个函数做的就是模拟真实的事件对象。

此外，还要配合 jQuery.event.fix 使用。 (2470 - 2527 行)

特殊事件都定义在 jQuery.event.special 。 (代码 2537 - 2567 行)

默认有 ready live beforeunload

一个特殊事件是 jQuery 内部特别处理的事件，它们可自定义这个事件如何绑定、添加、删除或触发。

代码 2292 - 2403 行，目的就是为了模拟触发某事件。这个函数支持模拟冒泡。

参考 http://www.cnblogs.com/rooney/archive/2008/12/03/1346449.html

9.内部的 Sizzle CSS Selector Engine (代码 3183 - 4518 行)

由于专门解释选择器的文章比较多，这里不多说了。

参考 http://www.cnblogs.com/rooney/archive/2008/12/02/1346135.html

http://www.cnblogs.com/rubylouvre/archive/2009/11/23/1607917.html

10.和节点有关扩展 (代码 4520 - 5492 行)

和节点有关的函数如: parent next 等这些查找 n 个节点的工具。

这些函数都依靠 jQuery.fn.nth() 通过 关系找到下个节点，如果这个节点是 Element, 则返回，否则继续找

由于这些函数都是比较易懂的，这里就不解释了。

一些 HACK 技巧强调下:

jQuery 1.5 的 clone 和 Mootools 的一样，采用 cloneFixAttributes

(代码 5196 - 5225 行)

cloneCopyEvent - 不是复制 data， 而是依据 data 重执行 bind

cloneFixAttributes - 先 clearAttributes ，然后 mergeAttributes (IE 专用) 这样可以保证属性正确拷贝。

这个函数用来按 HTML 返回节点， 就是 $("复杂 html ") 所使用的。

这里用到不常见的 document.createDocumentFragment();

如果是 <script> 则执行 jQuery.globalEval (全局执行)

为什么不直接把 <script> 放入 head ? 因为 单个 <script> 肯能导致泄漏。

删除元素内容子节点。

参考 http://www.cnblogs.com/70buluo/archive/2009/06/03/1495040.html

参考 http://www.cnblogs.com/rooney/archive/2008/12/02/1346135.html

11. 和样式有关扩展 (代码 5497 - 5825 行)

获取元素的 css 属性确实是很郁闷的事。jQuery 首先把一些属性单独处理， 其余的使用 style[name] || (IE ? getComputedStyle(elem, null).getProperty(name) : elem.currentStyle [name] 的方式获取，并稍微做点兼容处理。

具体可以参考 http://www.cnblogs.com/rubylouvre/archive/2009/09/05/1559883.html

http://www.cnblogs.com/rubylouvre/archive/2009/11/21/1607255.html

12.和 ajax 有关扩展 (代码 5830 - 7172 行)

jQuery 1.5 重写了 ajax 模块。

jQuery 的 ajax 其实也是使用一个模拟机制，而不是基于元素的 onreadystatechange

也就是说， jQuery 自己有个函数， 每隔 13ms 判断 readystate 是否 改了。

参考 http://www.cnblogs.com/qleelulu/archive/2008/04/21/1163021.html

13. 和效果有关扩展 (代码 7176 - 7696 行)

参考 http://www.cnblogs.com/rooney/archive/2008/12/03/1346475.html

附加说明下特效转换，

说到特效，很多人都知道这是依靠 setTimeout 完成的。但如何实现?

假设运动是 匀速的。从 x = x1 开始运动， 在 dt 中匀速运动到 x =x2 的位置。

而因为 需要在 dt 内把 x = x1 -> x2, 为了方便计算，特引入一个 delta， 满足 当前的 x = x1 + (x2 - x1) * delta 。

所以整个动画开始的时候 delta=0， 结束的时候则 dalta=1。这样，无论多大的 x1, x2 ，特效的计算都变为 delta 的计算。

delta 随时间变化，得到:

delta = f(t) (0 < t < dt) f 是一个映射。

每画一帧的时候 ， t 加 1， 根据 f 重新算 delta 和 x 。

假如我们需要一秒显示 fps 帧， 那就是说每 1000/fps 毫秒 显示一次， 也就说 每 1000/fps 毫秒就显示一帧，即 setTimeout (画帧, 1000/fps)。

总结下: 输入是 fps， x1， x2， dt 则代码:

```
var t = 0, f = 某个变换;

setInterval(function(){

 var delta = f(t);

 var x = x1 + (x2 - x1) * dalta;

 // 更新显示 x

 t++;

}, 1000 / fps);
```

然后介绍 f 是什么， f 输入是时间，输出 delta。f(0) = 0, f(dt) = 1。

在任一时刻: delta = (x - x1) / (x2 - x1)

当 x = 0 -> x

t = 0 -> dt

根据微分理解: f 是 x 方程的关于 t 的导函数。

当 x = k * t 为匀速运动，这里的 k = (x2 - x1) / dt

f = f|t = x' = k;

当 $x = k * t^2$ 是抛物线，

f = x' = 2 * k * t;

一般情况， 使用 sin 函数变换柔和点，所以 jquery 默认使用 sin 变换，所以变换函数是

x = sin(t - 3π/2) (最低点)

f = x' = -(cos(πt) - 1) / 2

(代码 7477 行)

14.和定位有关扩展 (代码 7700 - 8065 行)

jQuery.offset 用于管理定位，其它都是工具代码。

jQuery.offset.initialize 测试浏览器的定位法则。

jQuery.offse.bodyOffset 获取 body 的定位。因为这个定位和其它的不同，所有单独测试。

jQuery.fn.offset 这是核心定位函数， 返回绝对位置，新浏览器使用 getBoundingClientRect ，否则自己计算。

自己计算的方法就是遍历父元素，把相对位置相加，返回的就是和父元素的相对位置，那和 body 相对位置就是绝对位置。

参考 http://www.cnblogs.com/xuld/archive/2011/02/13/1953907.html

jQuery.fn.position jQuery.fn.position 返回和 style.left 一样。计算方法就是先 算 offset 和父节点(offsetParent) offset， 那么就可以得到偏移。

15. 总结

分析源码的时候可以对照代码理解。jQuery 虽然复杂但代码还是好读的。解读源码只要是为了更好地使用，所以知道源码后，就自然知道为什么 jQuery 可以做这么多事。

但 jQuery 始终不是唯一的框架，感叹它的设计同时，还应该看一下其它性格的框架，比如 Mootools 。