[【深入浅出jQuery】源码浅析--整体架构](http://www.cnblogs.com/coco1s/p/5261646.html)

<http://www.cnblogs.com/coco1s/p/5261646.html>

最近一直在研读 jQuery 源码，初看源码一头雾水毫无头绪，真正静下心来细看写的真是精妙，让你感叹代码之美。

其结构明晰，高内聚、低耦合，兼具优秀的性能与便利的扩展性，在浏览器的兼容性（功能缺陷、渐进增强）优雅的处理能力以及 Ajax 等方面周到而强大的定制功能无不令人惊叹。

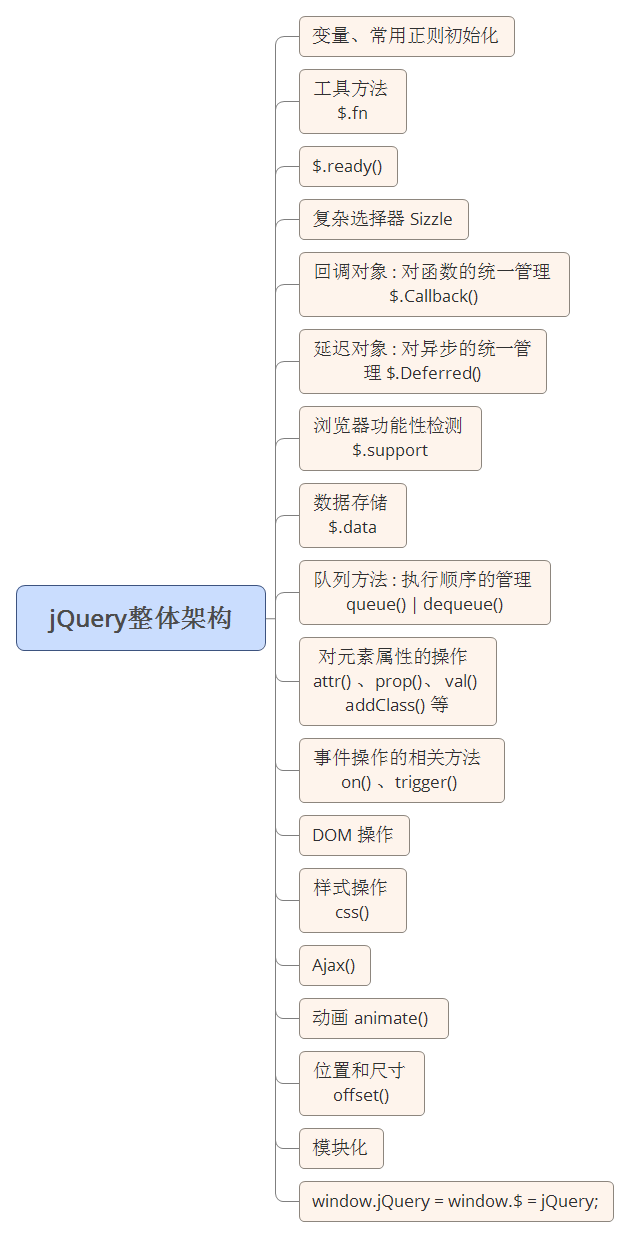
另外，阅读源码让我接触到了大量底层的知识。对原生JS 、框架设计、代码优化有了全新的认识，接下来将会写一系列关于 jQuery 解析的文章。

我在 github 上关于 jQuery 源码的全文注解，感兴趣的可以围观一下。[jQuery v1.10.2 源码注解](https://github.com/chokcoco/jQuery-)。

系列第二篇：[【深入浅出jQuery】源码浅析2--奇技淫巧](http://www.cnblogs.com/coco1s/p/5303041.html)

网上已经有很多解读 jQuery 源码的文章了，作为系列开篇的第一篇，思前想去起了个【深入浅出jQuery】的标题，资历尚浅，无法对 jQuery 分析的头头是道，但是 jQuery 源码当中确实有着大量巧妙的设计，不同层次水平的阅读者都能有收获，所以打算厚着脸皮将自己从中学到的一些知识点共享出来。打算从整体及分支，分章节剖析。本篇主要讲 jQuery 的整体架构及一些前期准备，先来看看 jQuery 的整体结构：

**jQuery 整体架构**



不同于 jQuery 代码各个模块细节实现的晦涩难懂，jQuery 整体框架的结构十分清晰，按代码行文大致分为如上图所示的模块。

初看 jQuery 源码可能很容易一头雾水，因为 9000 行的代码感觉没有尽头，所以了解作者的行文思路十分重要。

整体而言，我觉得 jQuery 采用的是**总--分**的结构，虽然JavaScript有着作用域的提升机制，但是 9000 多行的代码为了相互的关联性，并不代表所有的变量都要定义在最顶部。在 jQuery 中，只有全局都会用到的**变量、正则表达式**定义在了代码最开头，而每个模块一开始，又会定义一些只在本模块会使用到的变量、正则、方法等。所以在一开始的阅读的过程中会有很多看不懂其作用的变量，正则，方法。

所以，我觉得阅读源码很重要的一点是，摒弃面向过程的思维方式，不要刻意去追求从上至下每一句都要在一开始弄明白。很有可能一开始你在一个奇怪的方法或者变量处卡壳了，很想知道这个方法或变量的作用，然而可能它要到几千行处才被调用到。如果去追求这种逐字逐句弄清楚的方式，很有可能在碰壁几次之后阅读的积极性大受打击。

道理说了很多，接来下进入真正的正文，对 jQurey 的一些前期准备，小的细节进行分析：

**jQuery 闭包结构**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | // 用一个函数域包起来，就是所谓的沙箱  // 在这里边 var 定义的变量，属于这个函数域内的局部变量，避免污染全局  // 把当前沙箱需要的外部变量通过函数参数引入进来  // 只要保证参数对内提供的接口的一致性，你还可以随意替换传进来的这个参数  (function(window, undefined) {     // jQuery 代码  })(window); |

jQuery 具体的实现，都被包含在了一个立即执行函数构造的闭包里面，为了不污染全局作用域，只在后面暴露 $ 和 jQuery 这 2 个变量给外界，尽量的避开变量冲突。常用的还有另一种写法：

|  |  |
| --- | --- |
| 1  2  3 | (function(window) {     // JS代码  })(window, undefined); |

比较推崇的的第一种写法，也就是 jQuery 的写法。二者有何不同呢，当我们的代码运行在更早期的环境当中（pre-ES5，eg. Internet Explorer 8），undefined 仅是一个变量且它的值是可以被覆盖的。意味着你可以做这样的操作：

|  |  |
| --- | --- |
| 1  2 | undefined = 42  console.log(undefined) // 42 |

当使用第一种方式，可以确保你需要的 undefined 确实就是 undefined。

另外不得不提出的是，jQuery 在这里有一个针对压缩优化细节，使用第一种方式，在代码压缩的时候，window 和 undefined 都可以压缩为 1 个字母并且确保它们就是 window 和 undefined。

|  |  |
| --- | --- |
| 1  2  3  4  5 | // 压缩策略  // w -> windwow , u -> undefined  (function(w, u) {    })(window); |

**jQuery 无 new 构造**

 嘿，回想一下使用 jQuery 的时候，实例化一个 jQuery 对象的方法：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | // 无 new 构造  $('#test').text('Test');    // 当然也可以使用 new  var test = new $('#test');  test.text('Test'); |

大部分人使用 jQuery 的时候都是使用第一种无 new 的构造方式，直接 $('') 进行构造，这也是 jQuery 十分便捷的一个地方。当我们使用第一种无 new 构造方式的时候，其本质就是相当于 new jQuery()，那么在 jQuery 内部是如何实现的呢？看看：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | (function(window, undefined) {      var      // ...      jQuery = function(selector, context) {          // The jQuery object is actually just the init constructor 'enhanced'          // 看这里，实例化方法 jQuery() 实际上是调用了其拓展的原型方法 jQuery.fn.init          return new jQuery.fn.init(selector, context, rootjQuery);      },        // jQuery.prototype 即是 jQuery 的原型，挂载在上面的方法，即可让所有生成的 jQuery 对象使用      jQuery.fn = jQuery.prototype = {          // 实例化化方法，这个方法可以称作 jQuery 对象构造器          init: function(selector, context, rootjQuery) {              // ...          }      }      // 这一句很关键，也很绕      // jQuery 没有使用 new 运算符将 jQuery 实例化，而是直接调用其函数      // 要实现这样,那么 jQuery 就要看成一个类，且返回一个正确的实例      // 且实例还要能正确访问 jQuery 类原型上的属性与方法      // jQuery 的方式是通过原型传递解决问题，把 jQuery 的原型传递给jQuery.prototype.init.prototype      // 所以通过这个方法生成的实例 this 所指向的仍然是 jQuery.fn，所以能正确访问 jQuery 类原型上的属性与方法      jQuery.fn.init.prototype = jQuery.fn;    })(window); |

大部分人初看 jQuery.fn.init.prototype = jQuery.fn 这一句都会被卡主，很是不解。但是这句真的算是 jQuery 的绝妙之处。理解这几句很重要，分点解析一下：

1）首先要明确，使用 $('xxx') 这种实例化方式，其内部调用的是 return new jQuery.fn.init(selector, context, rootjQuery) 这一句话，也就是构造实例是交给了 jQuery.fn.init() 方法去完成。

2）将 jQuery.fn.init 的 prototype 属性设置为 jQuery.fn，那么使用 new jQuery.fn.init() 生成的对象的原型对象就是 jQuery.fn ，所以挂载到 jQuery.fn 上面的函数就相当于挂载到 jQuery.fn.init() 生成的 jQuery 对象上，所有使用 new jQuery.fn.init() 生成的对象也能够访问到 jQuery.fn 上的所有原型方法。

3）也就是实例化方法存在这么一个关系链

* jQuery.fn.init.prototype = jQuery.fn = jQuery.prototype ;
* new jQuery.fn.init() 相当于 new jQuery() ;
* jQuery() 返回的是 new jQuery.fn.init()，而 var obj = new jQuery()，所以这 2 者是相当的，所以我们可以无 new 实例化 jQuery 对象。

**jQuery 方法的重载**

jQuery 源码晦涩难读的另一个原因是，使用了大量的方法重载，但是用起来却很方便：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 获取 title 属性的值  $('#id').attr('title');  // 设置 title 属性的值  $('#id').attr('title','jQuery');    // 获取 css 某个属性的值  $('#id').css('title');  // 设置 css 某个属性的值  $('#id').css('width','200px'); |

方法的重载即是一个方法实现多种功能，经常又是 get 又是 set，虽然阅读起来十分不易，但是从实用性的角度考虑，这也是为什么 jQuery 如此受欢迎的原因，大多数人使用 jQuery() 构造方法使用的最多的就是直接实例化一个 jQuery 对象，但其实在它的内部实现中，有着 9 种不同的方法重载场景：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | // 接受一个字符串，其中包含了用于匹配元素集合的 CSS 选择器  jQuery([selector,[context]])  // 传入单个 DOM  jQuery(element)  // 传入 DOM 数组  jQuery(elementArray)  // 传入 JS 对象  jQuery(object)  // 传入 jQuery 对象  jQuery(jQuery object)  // 传入原始 HTML 的字符串来创建 DOM 元素  jQuery(html,[ownerDocument])  jQuery(html,[attributes])  // 传入空参数  jQuery()  // 绑定一个在 DOM 文档载入完成后执行的函数  jQuery(callback) |

所以读源码的时候，很重要的一点是结合 jQuery API 进行阅读，去了解方法重载了多少种功能，同时我想说的是，jQuery 源码有些方法的实现特别长且繁琐，因为 jQuery 本身作为一个通用性特别强的框架，一个方法兼容了许多情况，也允许用户传入各种不同的参数，导致内部处理的逻辑十分复杂，所以当解读一个方法的时候感觉到了明显的困难，尝试着跳出卡壳的那段代码本身，站在更高的维度去思考这些复杂的逻辑是为了处理或兼容什么，是否是重载，为什么要这样写，一定会有不一样的收获。其次，也是因为这个原因，jQuery 源码存在许多兼容低版本的 HACK 或者逻辑十分晦涩繁琐的代码片段，浏览器兼容这样的大坑极其容易让一个前端工程师不能学到编程的精髓，所以不要太执着于一些边角料，即使兼容性很重要，也应该适度学习理解，适可而止。

**jQuery.fn.extend 与 jQuery.extend**

extend 方法在 jQuery 中是一个很重要的方法，jQuey 内部用它来扩展静态方法或实例方法，而且我们开发 jQuery 插件开发的时候也会用到它。但是在内部，是存在 jQuery.fn.extend 和 jQuery.extend 两个 extend 方法的，而区分这两个 extend 方法是理解 jQuery 的很关键的一部分。先看结论：

1）**jQuery.extend(object)** 为扩展 jQuery 类本身，为类添加新的静态方法；

2）**jQuery.fn.extend(object)** 给 jQuery 对象添加实例方法，也就是通过这个 extend 添加的新方法，实例化的 jQuery 对象都能使用，因为它是挂载在 jQuery.fn 上的方法（上文有提到，jQuery.fn = jQuery.prototype ）。

它们的官方解释是：

1）jQuery.extend(): 把两个或者更多的对象合并到第一个当中，

2）jQuery.fn.extend()：把对象挂载到 jQuery 的 prototype 属性，来扩展一个新的 jQuery 实例方法。

也就是说，使用 jQuery.extend() 拓展的静态方法，我们可以直接使用 $.xxx 进行调用（xxx是拓展的方法名），

而使用 jQuery.fn.extend() 拓展的实例方法，需要使用 $().xxx 调用。

源码解析较长，点击下面可以展开，[也可以去这里阅读](https://github.com/chokcoco/jQuery-/blob/master/jquery-1.10.2__read.js)：

[+ View Code](http://www.cnblogs.com/coco1s/p/5261646.html)

需要注意的是这一句 jQuery.extend = jQuery.fn.extend = function() {} ，也就是 jQuery.extend 的实现和 jQuery.fn.extend 的实现共用了同一个方法，但是为什么能够实现不同的功能了，这就要归功于 Javascript 强大（怪异？）的 this 了。

1）在 jQuery.extend() 中，this 的指向是 jQuery 对象(或者说是 jQuery 类)，所以这里扩展在 jQuery 上；

2）在 jQuery.fn.extend() 中，this 的指向是 fn 对象，前面有提到 jQuery.fn = jQuery.prototype ，也就是这里增加的是原型方法，也就是对象方法。

**jQuery 的链式调用及回溯**

另一个让大家喜爱使用 jQuery 的原因是它的链式调用，这一点的实现其实很简单，只需要在要实现链式调用的方法的返回结果里，返回 this ，就能够实现链式调用了。

当然，除了链式调用，jQuery 甚至还允许回溯，看看：

|  |  |
| --- | --- |
| 1  2 | // 通过 end() 方法终止在当前链的最新过滤操作，返回上一个对象集合  $('div').eq(0).show().end().eq(1).hide(); |

当选择了 ('div').eq(0) 之后使用 end() 可以回溯到上一步选中的 jQuery 对象 $('div')，其内部实现其实是依靠添加了 prevObject 这个属性：



jQuery 完整的链式调用、增栈、回溯通过 return this 、 return this.pushStack() 、return this.prevObject 实现，看看源码实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | jQuery.fn = jQuery.prototype = {      // 将一个 DOM 元素集合加入到 jQuery 栈      // 此方法在 jQuery 的 DOM 操作中被频繁的使用, 如在 parent(), find(), filter() 中      // pushStack() 方法通过改变一个 jQuery 对象的 prevObject 属性来跟踪链式调用中前一个方法返回的 DOM 结果集合      // 当我们在链式调用 end() 方法后, 内部就返回当前 jQuery 对象的 prevObject 属性      pushStack: function(elems) {          // 构建一个新的jQuery对象，无参的 this.constructor()，只是返回引用this          // jQuery.merge 把 elems 节点合并到新的 jQuery 对象          // this.constructor 就是 jQuery 的构造函数 jQuery.fn.init，所以 this.constructor() 返回一个 jQuery 对象          // 由于 jQuery.merge 函数返回的对象是第二个函数附加到第一个上面，所以 ret 也是一个 jQuery 对象，这里可以解释为什么 pushStack 出入的 DOM 对象也可以用 CSS 方法进行操作          var ret = jQuery.merge(this.constructor(), elems);            // 给返回的新 jQuery 对象添加属性 prevObject          // 所以也就是为什么通过 prevObject 能取到上一个合集的引用了          ret.prevObject = this;          ret.context = this.context;            // Return the newly-formed element set          return ret;      },      // 回溯链式调用的上一个对象      end: function() {          // 回溯的关键是返回 prevObject 属性          // 而 prevObject 属性保存了上一步操作的 jQuery 对象集合          return this.prevObject || this.constructor(null);      },      // 取当前 jQuery 对象的第 i 个      eq: function(i) {          // jQuery 对象集合的长度          var len = this.length,              j = +i + (i < 0 ? len : 0);            // 利用 pushStack 返回          return this.pushStack(j >= 0 && j < len ? [this[j]] : []);      },  } |

总的来说，

1）end() 方法返回 prevObject 属性，这个属性记录了上一步操作的 jQuery 对象合集；

2）而 prevObject 属性由 pushStack() 方法生成，该方法将一个 DOM 元素集合加入到 jQuery 内部管理的一个栈中，通过改变 jQuery 对象的 prevObject 属性来跟踪链式调用中前一个方法返回的 DOM 结果集合

3）当我们在链式调用 end() 方法后，内部就返回当前 jQuery 对象的 prevObject 属性，完成回溯。

**jQuery 正则与细节优化**

不得不提 jQuery 在细节优化上做的很好。也存在很多值得学习的小技巧，下一篇将会以 jQuery 中的一些编程技巧为主题行文，这里就不再赘述。

然后想谈谈正则表达式，jQuery 当中用了大量的正则表达式，我觉得如果研读 jQuery ，正则水平一定能够大大提升，如果是个正则小白，我建议在阅读之前先去了解以下几点：

1）了解并尝试使用 Javascript 正则相关 API，包括了 test() 、replace() 、match() 、exec() 的用法；

2）区分上面 4 个方法，哪个是 RegExp 对象方法，哪个是 String 对象方法；

3）了解简单的零宽断言，了解什么是匹配但是不捕获以及匹配并且捕获。

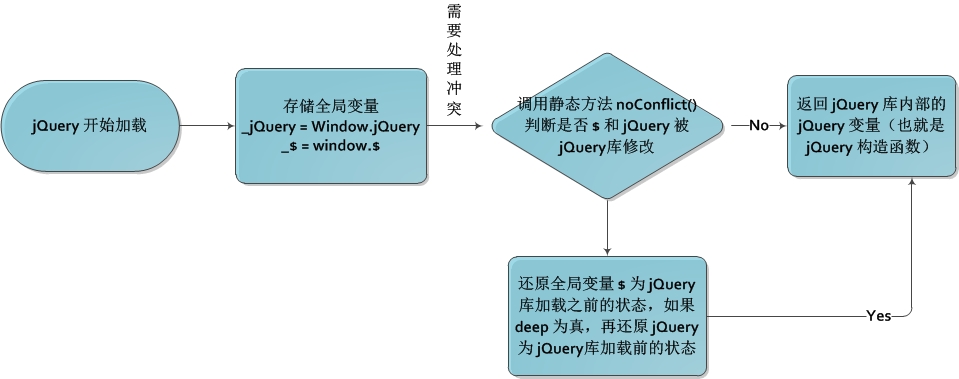
**jQuery 变量冲突处理**

最后想提一提 jQuery 变量的冲突处理，通过一开始保存全局变量的 window.jQuery 以及 windw.$ 。

当需要处理冲突的时候，调用静态方法 noConflict()，让出变量的控制权，源码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | (function(window, undefined) {      var          // Map over jQuery in case of overwrite          // 设置别名，通过两个私有变量映射了 window 环境下的 jQuery 和 $ 两个对象，以防止变量被强行覆盖          \_jQuery = window.jQuery,          \_$ = window.$;        jQuery.extend({          // noConflict() 方法让出变量 $ 的 jQuery 控制权，这样其他脚本就可以使用它了          // 通过全名替代简写的方式来使用 jQuery          // deep -- 布尔值，指示是否允许彻底将 jQuery 变量还原(移交 $ 引用的同时是否移交 jQuery 对象本身)          noConflict: function(deep) {              // 判断全局 $ 变量是否等于 jQuery 变量              // 如果等于，则重新还原全局变量 $ 为 jQuery 运行之前的变量（存储在内部变量 \_$ 中）              if (window.$ === jQuery) {                  // 此时 jQuery 别名 $ 失效                  window.$ = \_$;              }              // 当开启深度冲突处理并且全局变量 jQuery 等于内部 jQuery，则把全局 jQuery 还原成之前的状况              if (deep && window.jQuery === jQuery) {                  // 如果 deep 为 true，此时 jQuery 失效                  window.jQuery = \_jQuery;              }                // 这里返回的是 jQuery 库内部的 jQuery 构造函数（new jQuery.fn.init()）              // 像使用 $ 一样尽情使用它吧              return jQuery;          }      })  }(window) |

画了一幅简单的流程图帮助理解：



那么让出了这两个符号之后，是否就不能在我们的代码中使用 jQuery 或者呢 $ 呢？莫慌，还是可以使用的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | // 让出 jQuery 、$ 的控制权不代表不能使用 jQuery 和 $ ，方法如下：  var query = jQuery.noConflict(true);    (function($) {    // 插件或其他形式的代码，也可以将参数设为 jQuery  })(query);    //  ... 其他用 $ 作为别名的库的代码 |

**结束语**

对 jQuery 整体架构的一些解析就到这里，下一篇将会剖析一下 jQuery 中的一些优化小技巧，一些对编程有所提高的地方。

原创文章，文笔有限，才疏学浅，文中若有不正之处，万望告知。

如果本文对你有帮助，请点下推荐，写文章不容易。

系列第二篇：[【深入浅出jQuery】源码浅析2--奇技淫巧](http://www.cnblogs.com/coco1s/p/5303041.html)

最后，我在 github 上关于 jQuery 源码的全文注解，感兴趣的可以围观一下，给颗星星。[jQuery v1.10.2 源码注解](https://github.com/chokcoco/jQuery-)。