深入理解jQuery（1）——jQuery对象

Alex Sun
2014-10-04

一直在使用jQuery，但是对于其本质的机制却了解很少。这次通过查阅一些资料，以及查看jQuery的源码，有了一些较为深入的理解。该系列博文的参考资料为：
jQuery源码分析系列
逐行分析jQuery源码的奥秘

1. jQuery对象的创建

通常情况下，在面向对象的JS编程中，如果要实现一个对象，我们通常的做法是：

```
var Test = function() {

};

Test.prototype.someMethod = function() {
    // do something
};

var test = new Test();
test.someMethod();
```

然而我们在使用jQuery的时候，并没有使用new关键词来创建对象，也就是说，当我们调用jQuery()的时候，就创建了一个jQuery对象。我们可以看看jQuery源码中是如何实现的：

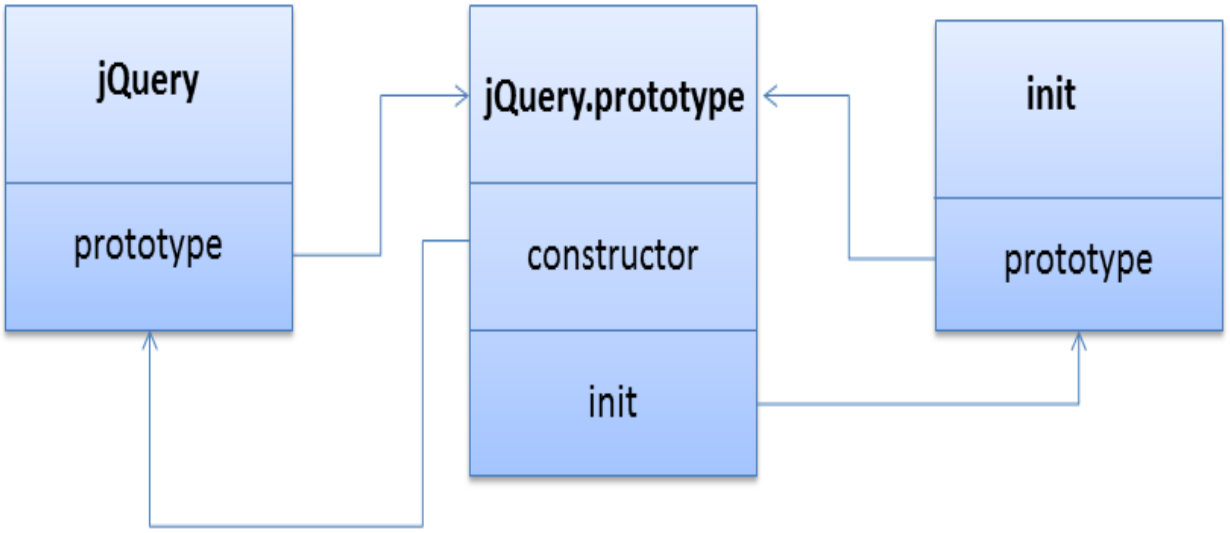
```
jQuery = function(selector, context) {
    return new jQuery.fn.init(selector, context, rootjQuery);
},

jQuery.fn = jQuery.prototype = {
    jquery: core_version,
    constructor: jQuery,
    init: function(selector, context, rootjQuery) {
        // return this
    },
```

```
    // other properties and methods
};

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

我们可以看到，在jQuery的构造函数中，返回的是一个init对象，其实可以理解为jQuery函数并非真正的构造函数，而init才是真正的构造函数，而jQuery函数只不过是在构造函数外的一层封装，用来得到一个实例对象。那么 `jQuery.fn.init.prototype = jQuery.fn` 是做什么的呢？因为init函数相当于是一个构造函数，那么我们通过这个构造函数生成的对象，肯定无法使用 `jQuery.prototype` 的原型方法，因此我们将其原型指针指过来，便可以使用jQuery的原型方法了。经过上面的一些构造，对象与原型的关系图如下：

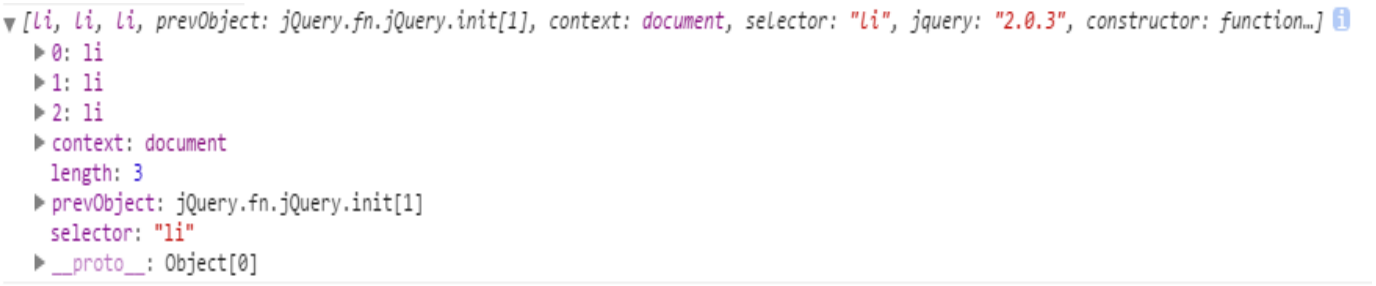


2. jQuery对象的结构

到此为止，我们已经知道了jQuery的构造函数及其原型之间的关系了。那么，一个jQuery对象的结构到底是怎样的呢，以如下代码为例：

```
$('li').css('background', 'red');
```

在这里，我们获取到了一个页面上的所有li元素，然后将其背景色设置为红色，通过查看，我们可以得到选择到的jQuery对象结构如下：



抛开图中的结构不谈，我们考虑一下，首先我们通过$('li')选择到的必定是一个类似数组的结构，然后在调用css()方法的时候，我们可以通过对得到的数据for循环，从而对每个元素依次设置样式。但是显然，我们看到的jQuery对象并不是一个数组，但是其结构非常巧妙，使用了类似数组的结构，对于选择到的每个元素，使用0、1、2等作为属性名，并且有一个length的属性，这样的话就可以通过类似数组循环的方法来对每个元素进行操作了。

一个jQuery对象中，每个元素存的都是原生的DOM对象，因此上面的代码其实相当于：

```
var oLi = $('li');
for (var i = 0; i < oLi.length; i++) {
    oLi[i].style.background = 'red';
}
```

那么接下来，原生的DOM对象和jQuery对象之间是什么关系呢。其实，jQuery对象中存储的即为DOM对象。如果我们得到了一个jQuery对象，那么我们可以通过下标或者get()方法来获得DOM对象，同样我们可以使用$()来将一个DOM对象转化为jQuery对象，例如：

```
var test = document.getElementById('test'); //DOM object
var $test = $('#test'); //jQuery object

$(test); //DOM object --> jQuery object
$test.get(0); //jQuery object --> DOM object

console.log(test === $test.get(0)); //true
```

3. jQuery原型探究

```
jQuery.fn = jQuery.prototype = {
    // ......
}
```

上面的代码片段定义了jQuery的原型，其中jQuery.fn为原型的别名。在原型中，定义了一些属性和方法，下面一一解释：

- jquery：版本号，例如可以通过$().jquery得到
- constructor：指向构造函数的指针，因为这里是直接将jQuery.prototype指向了一个新的对象，所以需要显示声明该指针
- init()：真正的构造方法，后面将会介绍
- selector：选择器，例如使用$(“#header p”)，那么selector就是”#header p”
- length：选择到的元素的个数
- toArray()：将一个jQuery对象转化为数组，数组的每一项都是一个原生DOM对象
- get()：获得指定下标的DOM对象，如果未指定参数，那么相当于直接调用toArray()方法
- pushStack()：压入调用栈，后面将会详细介绍
- each()：类似于数组的forEach方法，其本质是调用了$.each()方法
- ready()：例如$(document).ready(function(){})，将在后面进行介绍
- slice()：类似于数组的slice方法
- first()：获取第一个元素
- last()：获取最后一个元素
- eq()：获取指定下标的元素，若参数为负数，则从后向前进行查找
- map()：类似于数组的map方法
- end()：结束当前调用，返回调用栈中的上一个对象
- push()：类似数组的push方法
- sort()：类似数组的sort方法
- splice()：类似数组的splice方法

这里需要明确区分开jQuery对象和DOM对象。上面的方法中，toArray()和get()返回的都是原生的DOM对象，而slice()、first()、last()、eq()方法都是获取到需要的元素，然后将这些元素封装成一个新的jQuery对象，然后压入调用栈；map()方法也会创建新的jQuery对象并压入调用栈。另外源码注释里也说明了push()、sort()和splice()仅仅是作为内部方法来使用。

4. 调用栈

在分析调用栈之前我们先来看一个例子，考虑如下的代码片段：

```
<body>
    <ul>
        <li>hello world</li>
        <li>hello world</li>
        <li>hello world</li>
    </ul>
</body>
```

假如我们希望整个列表的背景为红色而列表的第二项背景为蓝色，我们可以这样做：

```
$(function(){
    $("ul").css("background","red");
```

```
    $("ul li:eq(1)").css("background","blue");
});
```

也可以通过链式操作这样做：

```
$(function(){

$("ul").css("background","red").find("li:eq(1)").css("background","blue"
);
});
```

但是，假如我们需要先改变列表第二项的背景色，再改变列表的背景色，该如何做呢。当然，我们可以将第一种做法的两行代码颠倒一下就可以了，但是能不能通过链式操作使用一句代码完成呢？这个时候便需要用到调用栈了，首先我们来看一下如何实现：

```
$(function(){

$("ul").find("li:eq(1)").css("background","blue").end().css("background"
,"red");
});
```

这里我们使用了 end() 方法，该方法的作用是结束当前调用，返回调用栈的上一个对象。其实原理很简单：当执行 $("ul") 的时候，会生成一个jQuery对象，然后在执行 find("li:eq(1)") 的时候，又会生成一个新的jQuery对象。我们可以假设有一个栈，将这两个jQuery对象依次入栈，然后当执行 css("background","blue") 的时候，是作用于栈顶元素的。当执行 end() 的时候，就执行一次出栈操作。那么jQuery究竟是如何实现的呢，我们来看 pushStack() 的源码：

```
// Take an array of elements and push it onto the stack
// (returning the new matched element set)
pushStack: function( elems ) {

    // Build a new jQuery matched element set
    var ret = jQuery.merge( this.constructor(), elems );

    // Add the old object onto the stack (as a reference)
    ret.prevObject = this;
    ret.context = this.context;

    // Return the newly-formed element set
    return ret;
}
```

可以发现并没有真正的栈，而是每次执行pushStack的时候，就将新的jQuery对象的prevObject指针

指向当前对象，那么就形成了一个链表，当执行end()的时候只需要返回当前jQuery对象的prevObject就可以了。

5. 深入init

init()其实是真正的jQuery构造函数，其结构如下：

```
function( selector, context, rootjQuery ) {
    // ......
}
```

然而我们知道，$()函数的使用方法有多种，可以用来选择元素，也可以用来创建对象。总的来说，其用法大致分为如下几种：

- 等待DOM准备好执行：$(function(){})
- 选择元素：简单元素选择器例如 $(“div”)、$(“#header”)；复杂元素选择器如$(“.sidebar > div ul”)
- 创建元素：例如$(“ ”)等

下面将对这些情况进行具体分析。init()的大体结构如下所示：

```
function( selector, context, rootjQuery ) {
    // ......
    // HANDLE: $(""), $(null), $(undefined), $(false)
    if ( !selector ) {
        return this;
    }

    // Handle HTML strings
    if ( typeof selector === "string" ) {
        // ......

        // Match html or make sure no context is specified for #id
        if ( match && (match[1] || !context) ) {
            // HANDLE: $(html) -> $(array)
            if ( match[1] ) {
                // ......
                // HANDLE: $(html, props)
                if ( rsingleTag.test( match[1] ) &&
jQuery.isPlainObject( context ) ) {
                    // ......
                }
                return this;
```

```
                // HANDLE: $(#id)
                } else {
                    elem = document.getElementById( match[2] );
                    // ......
                    return this;
                }

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

            // HANDLE: $(expr, context)
            // (which is just equivalent to: $(context).find(expr)
            } else {
                return this.constructor( context ).find( selector );
            }

        // HANDLE: $(DOMElement)
        } else if ( selector.nodeType ) {
            // ......
            return this;

        // HANDLE: $(function)
        // Shortcut for document ready
        } else if ( jQuery.isFunction( selector ) ) {
            return rootjQuery.ready( selector );
        }

        // ......
        // $([]), $({})
        return jQuery.makeArray( selector, this );
    }
```

首先是根据selector来判断，先过滤掉 `$("")` 、 `$(null)` 、 `$(undefined)` 、 `$(false)` 等这些无效的选择，然后根据selector的数据类型进行判断，如果selector是字符串，说明是选择元素或者创建元素；如果selector是一个DOM节点，例如 `$(this)` 、 `$(document)` 等，那么就用这个DOM元素来创建jQuery对象即可；如果selector是一个函数，即 `$(function(){})` ，则转为 `$(document).ready(function(){})` ，由此我们可以发现这两种写法本质上是完全一样的；最后是针对例如 `$([])` 、 `$({})` 等。

当selector是字符串的时候，首先判断是不是创建元素或者选择ID，如果是则进入if语句 `if(match[1] || !context)` ，否则根据其context使用 `find()` 进行选择。 `find()` 的本质是使用了Sizzle，关于Sizzle在这里暂不介绍，只需了解它是jQuery集成的一个函数库，用来实现复杂的选择器即可。

这样一来jQuery()函数就将各种功能统一了起来，对外是简洁的接口，而内部实现确实非常复杂

的。至于代码的详细分析，这里不做赘述。