**jQuery中的**[**$符号到底是什么----详解jQuery的$符号和init函数**](http://blog.csdn.net/pgmsoul/article/details/8002805)

jQuery是现在最流行的[JavaScript](http://lib.csdn.net/base/javascript)框架， $是其中最常见的符号，那么$到底是什么东西？它可以接受一个字符，也可以接受一个文档对象，亦或者一个函数，也可以调用一个函数。$符号作为元素选择器的简写，最早是由Prototype库使用，来简写getElementById，jQuery沿袭这一理念，并发扬光大，使$符号成为了jQuery最别具一格的特点。熟悉jQuery的人应该知道，几乎jQuery所有操作，都是从$符号开始，当作为元素选择器的时候，操作结果返回的是一个jQuery对象。如下代码，即为jQuery类的构造函数的主要代码

<!DOCTYPE **html**>  
<**html lang="en"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>Title</**title**>  
</**head**>  
<**body**>  
<**script**>  
 **var *jQuery*** = (**function**() {  
 *//创建jQuery对象,给所有的jQuery方法提供统一的入口,避免繁琐难记* **var** *jQuery* = **function**( selector, context ) {  
 *//jQuery的构造对象,调用了jQuery.fn.init方法  
 //最后返回jQuery.fn.init的对象* **return new** *jQuery*.**fn**.init( selector, context, rootjQuery );  
 },  
  
 .....  
  
 *//定义jQuery的原型,jQuery.fn指向jQuery.prototype对象  
 jQuery*.**fn** = *jQuery*.**prototype** = {  
 *//重新指定构造函数属性,因为默认指向jQuery.fn.init* constructor: *jQuery*,  
 init: **function**( selector, context, rootjQuery ) {.....},  
  
 ......  
  
 }  
  
 ......  
  
 *//返回jQuery变量,同时定义将全局变量window.jQuery和window.$指向jQuery* **return** (**window**.**jQuery** = **window**.$ = *jQuery*);  
  
 })();  
</**script**>  
</**body**>  
</**html**>

从以上jQuery的主体结构，我们可以看出，当首次执行完毕后，全局变量$和jQuery，都是指向了var jQuery=function（selector，context）{}这个函数，这里，就可以下个结论，$就是jQuery的别名，实际调用jQuery.fn.init。

再看看var jQuery=function（selector，context）{}这个构造函数，为什么里面不直接返回jQuery的对象？而是调用另外一个方法呢？

假如直接返回对象的话，每次使用jQuery对象，都要new jQuery() 这样的话，十分不方便，直接将new 这个操作封装在jQuery构造函数里面，简化了实例化的操作，同时，jQuery通过jQuery或者$符号，统一了接口，方便代码的编写，化繁为简，提高效率。

那么jQuery类具体是如何构造的？居然能支持各种参数形式的调用？  
先看jQuery.fn.init的源码，jQuery的真实构造器。

<!DOCTYPE **html**>  
<**html lang="en"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>Title</**title**>  
</**head**>  
<**body**>  
<**script**>  
 */\*所有查找或生成元素的结果，封装为jQuery对象数组返回.   
 \*/* init: *function*( selector, context, rootjQuery ) {  
 **var** match, elem, ret, doc;  
  
 *// 1)处理 $(""), $(null), or $(undefined)   
 //this指向jQuery对象* **if** ( !selector ) {  
 **return this**;  
 }  
  
 *// 2)处理 $(DOMElement)   
 //selector.nodeType得知为DOM元素,如果是DOM元素直接放进jQuery对象数组中* **if** ( selector.**nodeType** ) {  
 **this**.**context** = **this**[0] = selector;  
 **this**.**length** = 1;  
 **return this**;  
 }  
  
 *//3)body元素只出现一次, 优化查找* **if** ( selector === **"body"** && !context && **document**.**body** ) {  
 **this**.**context** = **document**;  
 **this**[0] = **document**.**body**;  
 **this**.**selector** = **"body"**;  
 **this**.**length** = 1;  
 **return this**;  
 }  
  
 *//4)如果是字符串,有六种情况,   
 /\*   
 \*(1)单个html元素 不带属性对象字面量 ：createElement + merge   
 \*(2)单个html元素 带属性对象字面量 ：createElement + attr + merge   
 \*(3)多个html元素 ：buildFragment + merge   
 \*(4)#id 不带context ：getElementById或者getElementById + Sizzle   
 \*(5)#id 带context ：Sizzle   
 \*(6)experession string ：Sizzle   
 \*(7)标签选择器 ：Sizzle（内置getElementByTagName）   
 \*/* **if** ( typeof selector === **"string"** ) {  
 *// 判断是否为HTML string 还是 ID   
 //如果是HTML strings match[1] 非空   
 //如果是ID strings match[1] 空   
 //quickExpr = /^(?:[^<]\*(<[\w\W]+>)[^>]\*$|#([\w\-]+)$)/,* match = quickExpr.exec( selector );  
  
 *// 分析匹配结果，且当#id没有context参数，例如不是$('#xxx',xxx)* **if** ( match && (match[1] || !context) ) {  
  
 *// 处理HTML字符 $(html) -> $(array)* **if** ( match[1] ) {  
 *//如果context为jQuery对象，则取用第一个元素，即是context[0]* context = context instanceof jQuery ? context[0] : context;  
 *//取得document文档* doc = (context ? context.**ownerDocument** || context : **document**);  
  
 *//判断是否为单个元素字符串* ret = rsingleTag.exec( selector );  
 *//单个元素* **if** ( ret ) {  
 *//带对象属性字面量   
 //检查context是否为对象字面量,适用场景   
 //例如$('<div>', { 'id': 'test', 'class': 'test' });* **if** ( jQuery.isPlainObject( context ) ) {  
 selector = [ **document**.createElement( ret[1] ) ];  
 jQuery.fn.attr.call( selector, context, **true** );  
  
 } **else** {  
 *//不带对象字面量   
 //例如$('<div>')* selector = [ doc.createElement( ret[1] ) ];  
 }  
  
 } **else** {  
 *//如果是多个元素字符串，例如$('<div><a></a></div>')* ret = jQuery.buildFragment( [ match[1] ], [ doc ] );  
 selector = (ret.cacheable ? jQuery.clone(ret.fragment) : ret.fragment).**childNodes**;  
 }  
 *//将生成结果selector 合并到jQuery对象中* **return** jQuery.merge( **this**, selector );  
  
 *// 处理$("#id"),例如$("#xxx");* } **else** {  
 elem = **document**.getElementById( match[2] );  
  
 **if** ( elem && elem.**parentNode** ) {  
 *//处理IE和Opera ID 与 Name 混淆的bug,使用Sizzle查找* **if** ( elem.**id** !== match[2] ) {  
 **return** rootjQuery.find( selector );  
 }  
  
 *// 否则，简单插入jQuery对象数组* **this**.**length** = 1;  
 **this**[0] = elem;  
 }  
  
 **this**.**context** = **document**;  
 **this**.**selector** = selector;  
 **return this**;  
 }  
  
 *// 处理 $(expr, $(...))，使用Sizzle查找，例如$("div"),$('div > a'),$('div,a'),$('div:first')* } **else if** ( !context || context.jquery ) {  
 **return** (context || rootjQuery).find( selector );  
  
 *// 处理: $(expr, context),例如$('div a');或者$('a','div')或者$('div').find('a');* } **else** {  
 **return this**.**constructor**( context ).find( selector );  
 }  
  
 *//5)处理: $(function),设置DOM载的时候绑定的函数,等同于$().ready(){foo}* } **else if** ( jQuery.isFunction( selector ) ) {  
 **return** rootjQuery.ready( selector );  
 }  
 *//6)处理：$($(...))，完成克隆jQuery对象的简单参数，具体由makeArray完成* **if** (selector.**selector** !== **undefined**) 完成加{  
 **this**.**selector** = selector.**selector**;  
 **this**.**context** = selector.**context**;  
 }  
 *//使用makeArray，为jQuery对象添加元素,例如$([1,2]);* **return** jQuery.makeArray( selector, **this** );  
 },  
</**script**>  
</**body**>  
</**html**>

总结

从源码可以看出，jQuery 通过各种条件判断和强大的正则表达式，实现了各种参数的调用。而$符号，其实是对jQuery类构造函数的引用，此函数实际调用了jQuery.fn.init(即是jQuery.prototype.init)来生成jQuery对象,其中jQuery.prototype的所有方法都被jQuery的对象继承。$.func实际是jQuery类的静态方法，所以$即是jQuery类的构造函数，支持各种条件的查找和生成并返回DOM对象构成jQuery对象，同时也是一个类，是所有jQuery静态方法的入口。

[**jQuery源码中的“new jQuery.fn.init()”什么意思？**](http://www.cnblogs.com/SheilaSun/p/4779895.html)**（某知名博主博客引入，纯原文摘抄，供大家参考）**

**引子**

最近打算试试看看jQuery的源码，刚开个头就卡住了。无论如何都理解不了jQuery源码入口部分中的

**return new** jQuery.fn.init( selector, context )，  
下面一点点地说下我对jQuery入口源码的理解。

**自执行的匿名函数**

jQuery源码最外层的结构如下:

<**script**>  
 (**function**(window,undefined){  
 ...  
 })(**window**);  
</**script**>

任何库的引入都得做到不污染全局变量，得有自己的命名空间。上面的自执行匿名函数就可以做到这点，把所有库私有的变量和方法，都包到一个私有的空间内，允许外界访问的属性或方法可以挂载到window上。

例如下面这段代码：

<**script**>  
 (**function**(){  
 **var** count=0;  
 **var** *addOne*=**function**(){  
 alert(count++);  
 };  
 **window**.outerAddOne=*addOne*; *//挂到window上外界方可访问* })();  
  
 outerAddOne();*//alert "0"* **console**.log(count);*//error* **console**.log(addOne);*//error*</**script**>

内部定义的count变量以及addOne方法，外部环境下是无法访问到的，但是在window上挂载一个方法outerAddOne，指向addOne，外界就可以访问到了。

OK，了解了这个自执行匿名函数的作用，这里还有两个问题。

**第一，为什么要传入window？**

看了上面的outerAddOne这个例子，就会发现，不传入window也没什么嘛，照样可以把方法挂到window身上啊。  
两个原因：

**首先，从代码压缩混淆的角度考虑。**

我们用线上工具来压缩混淆下面这段示例代码：

<**script**>  
 **function** *say*(){  
 **var** name=**"naima"**;  
 **window**.**description**=**"hi "**+name;  
 }  
</**script**>

压完混完后瘦了一点：

<**script**>  
 **function** *say*(){**var** a=**"naima"**;**window**.**description**=**"hi "**+a}  
</**script**>

看到没有，用a代替了name，但是window既不是声明的局部变量也不是参数，是不会被压缩混淆的，所以将window作为参数传入可解决这个问题。

**其次，传入window参数，就可以不用沿着作用域链一层层向上查找直到顶层作用域去获取window对象了，访问更快了。**

**第二，为什么要传入undefined？**

undefined并不是JS中的关键字，在IE8及以下中是可以对其重新赋值的。

<**script**>  
 **var *undefined***=**"new value"**;  
 alert(***undefined***);*//alert “new value"*</**script**>

在参数列表中给出undefined参数，但是不传入值，那么这个参数值就是undefined值了。

**jQuery对象的构建**

先看jQuery源码中如何对jQuery赋值的：

<**script**>  
 jQuery = **function**( selector, context ) {  
 *// The jQuery object is actually just the init constructor 'enhanced'* **return new** jQuery.fn.*init*( selector, context, rootjQuery );  
 }  
  
</**script**>

我就是被new jQuery.fn.init()这里弄晕了，先在这里暂停，回想一下平常我是如何使用jQuery的($即对应‘jQuery')：

<**script**>  
 $(**'body'**).css(**'background'**,**'red'**);  
 $.parseJSON(**'{}'**);  
</**script**>

要实现这两种调用，$('body')应该是一个实例对象，css是每个实例共享的方法，是原型上的方法。而$则是一个类，parseJSON则是类的静态方法。  
接下来，我们试着往这个结果上靠。

**如何不用new关键字得到jQuery对象？**

回想一下平常我都是怎么构建实例对象的，通常我会这样写一个Prince类：

<**script**>  
 **function** *Prince*(name){  
 **this**.**name**=name;  
 **this**.**body**=**"human"**;  
 }  
  
 *Prince*.**prototype**.change=**function**(){  
 **this**.**body**=**"frog"**;  
 };  
</**script**>

然后我会这样去获取一个Prince实例对象：

<**script**>  
 **var *prince***=**new** Prince(**"Harry"**);  
 ***prince***.change();  
</**script**>

如果我年纪大了忘记用new关键字了，程序就报错了：

<**script**>  
 **var *a***=Prince(**'harry'**);  
 ***a***.change();*//error,"Cannot read property 'change' of undefined"*</**script**>

除了调用方法会出错之外，window还被挂载了两个变量上去，何其无辜。

但是获取jQuery对象（以下简称JQ对象）用new和不用new都可以，返回的是一样样的。

<**script**>  
 **console**.log($(**'\*'**).**length**);*//14* **console**.log(**new** $(**'\*'**).**length**);*//14*</**script**>

为了做到这点，我们很容易想到需要在构造函数内部返回对象。引用下我在另一篇博文“[JavaScript中的普通函数与构造函数](http://www.cnblogs.com/SheilaSun/p/4398881.html)”里写的：

构造函数有return值怎么办？  
构造函数里没有显式调用return时，默认是返回this对象，也就是新创建的实例对象。  
当构造函数里调用return时，分两种情况：  
1.return的是五种简单数据类型：String，Number，Boolean，Null，Undefined。  
这种情况下，忽视return值，依然返回this对象。  
2.return的是Object  
这种情况下，不再返回this对象，而是返回return语句的返回值。

所以我们应该在jQuery构造函数内部去返回一个对象，这样就可以不用new的方式去创建JQ对象了，其实这时候，构造函数就相当于一个工厂函数了。  
那么核心问题来了。

**该返回什么样的对象？对于这个对象有何要求？**

**这个对象必须可以调用jQuery.prototype上的方法。**

我们使用或自己写jQuery插件的时候会经常遇到$.fn这个对象，很多插件都是通过扩展这个对象来实现的。  
$.fn其实对应着jQuery.prototype，$和fn分别是jQuery和prototype的简写方式，只要我们把方法扩展到这个原型对象身上，通过$()获取的JQ对象都是可以访问到方法的。  
例如：

<**script**>  
 $.fn.greeting=**function**(){alert(**'hi'**)};  
 $(**'body'**).greeting();*//alert 'hi'*</**script**>

所以，工厂函数内部返回的对象一定要可以调用jQuery.prototype上的方法。

是时候看John Resig到底是怎么做的啦。

**jQuery源码**

<**script**>  
 jQuery = **function**( selector, context ) {  
 **return new** jQuery.**fn**.**init**( selector, context, rootjQuery );  
 },  
 jQuery.**fn** = jQuery.**prototype** = { *//fn即对应prototype* constructor: jQuery,  
 **init**: **function**( selector, context, rootjQuery ) {  
 ...  
 **return this**;  
 }  
 ...  
 }  
 jQuery.**fn**.**init**.**prototype** = jQuery.**fn**;  
</**script**>

在chrome里调试时候添加JQ对象的watch，会看到类似如下的结果：

$('\*'): n.fn.init[14]

看到上面这段源码，原因就很明显了，其实我们所说的JQ对象根本就是init函数的实例对象，而init则是jQuery原型上的一个对象，它本身是没有什么方法的，全靠从jQuery原型上拿。

"jQuery.fn.init.prototype = jQuery.fn"这句很重要，它将init的原型指向jQuery的原型，所以JQ对象才可以访问‘css'、'show'、'hide'这些写在jQuery.fn上的方法。

我们可能会有疑问，为何要从init这绕这么一大圈来访问jQuery的原型，而不是直接返回一个jQuery实例直接通过这个实例来访问自身原型？比如说代码可以写成这样：

<**script**>  
 jQuery = **function**( selector, context ) {  
 **return new** jQuery();  
 }  
</**script**>

问题很明显，这样做只会大家一起死，死在循环里。

好，那我接受init的存在，但是我这样写难道不可以吗？

<**script**>  
 jQuery = **function**( selector, context ) {  
 **return** jQuery.fn.*init*();*//不同点在于去掉了new关键字* }  
</**script**>

让我们做点动作来证明加上new是有用的。

<**script**>  
 jQuery = **function**( selector, context ) {  
 **return** jQuery.**fn**.init();  
 },  
 jQuery.**fn** = jQuery.**prototype** = {  
 init: **function**() {  
 **this**.**name**=**'sheila'**;  
 **return this**;  
 },  
 **anotherName**:**'sunwukong'** };  
 **var *jq***=jQuery();  
 **console**.log(***jq***.**anotherName**);*//"sunwukong"* **console**.log(***jq***.name);*//"sheila"*</**script**>

上面这段代码是为了说明this的作用域问题，其不仅能访问init函数内部，还能向上一层到fn对象。我听人家说，做框架的，作用域要独立才好呢。  
给它加上new关键字：

<**script**>  
 ...  
 **return new** jQuery.fn.*init*();  
 ...  
  
 **console**.log(jq.anotherName);*//undefined* **console**.log(jq.name);*//"sheila"*</**script**>

这样this的作用域就独立出来了。

经博友评论提醒，加不加new还牵涉到一个更重要的问题：返回的对象究竟是谁。不加new的情况下，'jQuery.fn.init()'相当于调用方法，this指向的以及最后返回的都是同一个jQuery.fn对象，$('body')和$('p')就没有区分了。显然，这是不合理的。而加了new，就是每次用构造函数实例化了一个新对象，彼此都是不同的。

有任何不妥之处或错误欢迎各位指出，不胜感激~

**jquery中如何将数组转化为json字符串，然后再转化回来？**

在开发的过程中，如果对于少量参数的前后台传递，可以直接采用ajax的data函数，按json格式传递，后台Request即可，但有的时候，需要传递多个参数，这样后台

接受的时候Request多个很麻烦，此时要按照类的格式或者 集合的形式进行传递。

例如：前台按类的格式传递JSON对象：

<**script**>  
 **var *jsonUserInfo*** = **"{\"TUserName\":\""** + userName + **"\",\"TInterest\":\""** + interest + **"\",\"TSex\":\""** + sex + **"\",\"TCity\":\""** + city + **"\",\"TDetail\":\""** + detail + **"\"}"**;  
</**script**>

如拼出的jsonUserInfo 无转义符号，需要var jsonArrayFinal = JSON.stringify(jsonArray);进行转换后再传递。

代码如下：

<**script**>  
 $.ajax(  
 {  
 **type**: **"post"**,  
 **url**: **"ReceiveHandler1.ashx"**,  
 **data**: { **userInfo**: jsonUserInfo, **flag**: **"123456"**, **key**: **"654321"** },  
 **dataType**: **"text"**,  
 success: **function**(data) {  
 $(**"#divShow"**).html(data);  
 }  
 });  
</**script**>

如果前台传递多个类格式的JSON数组，也就是集合类型：

例如：

[{**"name"**:**"a"**},{**"name"**,**"b"**},{**"name"**,**"c"**}]

则无法传递，此时必须使用JSON.stringify将数组对象转换成字符串，再进行AJAX传递即可。

比如我有两个变量，我要将a转换成字符串，将b转换成JSON对象：

**var *a***={**"name"**:**"tom"**,**"sex"**:**"男"**,**"age"**:**"24"**};  
**var *b***=**'{"name":"Mike","sex":"女","age":"29"}'**;

在Firefox，chrome，opera，safari，ie9，ie8等高级浏览器直接可以用JSON对象的stringify()和parse()方法。

JSON.stringify(obj)将JSON转为字符串。JSON.parse(string)将字符串转为JSON格式；

上面的转换可以这么写：

**var *a***={**"name"**:**"tom"**,**"sex"**:**"男"**,**"age"**:**"24"**};  
**var *b***=**'{"name":"Mike","sex":"女","age":"29"}'**;  
**var *aToStr***=**JSON**.stringify(***a***);  
**var *bToObj***=**JSON**.parse(***b***);  
alert(typeof(***aToStr***)); *//string*alert(typeof(***bToObj***));*//object***JSON**.stringify();

ie8(兼容模式),ie7和ie6没有JSON对象，不过<http://www.json.org/js.html>提供了一个json.js,这样ie8(兼容模式),ie7和ie6就可以支持JSON对象以及其stringify()和parse()方法；你可以在https://github.com/douglascrockford/JSON-js上获取到这个js，一般现在用json2.js。

ie8(兼容模式),ie7和ie6可以使用eval()将字符串转为JSON对象，

**var *c***=**'{"name":"Mike","sex":"女","age":"29"}'**;  
**var *cToObj***=eval(**"("**+***c***+**")"**);  
alert(typeof(***cToObj***));

jQuery中也有将字符串转为JSON格式的方法jQuery.parseJSON( json )，接受一个标准格式的 JSON 字符串，并返回解析后的 JavaScript （JSON）对象。当然如果有兴趣可以自己封装一个jQuery扩展，jQuery.stringifyJSON(obj)将JSON转为字符串。

[**jQuery绑定事件的四种方式**](http://www.cnblogs.com/lipenghui/p/6014669.html)

jQuery中提供了四种事件监听方式，分别是bind、live、delegate、on，对应的解除监听的函数分别是unbind、die、undelegate、off。在讲解之前补充下基本知识：DOM树。我们的页面可以理解为一棵DOM树，当我们在叶子结点上做什么事情的时候（如click一个a元素），如果我们没有人为的设置stopPropagation(Moder Browser), cancelBubble(IE), //阻止传播（现代的浏览器），取消冒泡（ie浏览器），那么它的所有父元素，祖宗元素都会受之影响，绑定的事件也对它们产生作用。

**示例：$('a').bind('click', function() { alert("That tickles!") });**

下面分别介绍jQuery中绑定事件的四种方式。

1. Bind

$(selector).bind(event,data,function)

　　event：事件，必选，一个或多个事件；　　data：参数，可选；　　fn：绑定事件发生时执行的函数，必选。

　　bind()是最直接的、存在最久的绑定方法。

　　优点：这个方法提供了一种在不同浏览器中对事件处理的兼容性解决方案。

缺点：它会绑定事件到所有的目标元素上，有一个绑一个，在页面上的元素不会动态添加的时候使用它没什么问题。但如果列表中动态增加一个“列表元素5”，点击它是没有反应的，必须再bind一次才行。也就是说，它不会绑定到在它执行完后动态添加的那些元素上。当元素很多时，会出现效率问题。当页面加载完的时候，你才可以进行bind()，所以可能产生效率问题。

1. Live

$(selector).live(event,data,function)

　　这个方法用到了事件委托的概念来处理事件的绑定。

　　优点：这里仅有一次的事件绑定，绑定到document上而不像.bind()那样给所有的元素挨个绑定。 那些动态添加的元素依然可以触发那些早先绑定的事件，因为事件真正的绑定是在document上。你可以在document ready之前就可以绑定那些需要的事件。

1. Delegate

$(selector).delegate(childSelector,event,data,function)

childSelector：selector的子元素，必需项。

　　delegate()有点像live(),不同于.live()的地方在于，它不会把所有的event全部绑定到document,而是由你决定把它放在哪儿。

　　优点：你可以选择你把这个事件放到那个元素上了。需要迭代查找所有的selector/event data来决定那个子元素来匹配，但是因为你可以决定放在那个根元素上，所以可以有效的减小你所要查找的元素。可以用在动态添加的元素上。

　　缺点：需要查找绑定的子元素，尽管比document少很多了，不过，你还是得浪费时间来查找。

1. on

其实.bind(), .live(), .delegate()都是通过.on()来实现的，.unbind(), .die(), .undelegate(),也是一样的都是通过.off()来实现的，

优点：提供了一种统一绑定事件的方法。

缺点：也许会对你产生一些困扰，因为它隐藏了前面我们所介绍的三种方法的细节。

五、四种方式的异同和优缺点

　　相同点：1.都支持单元素多事件的绑定；空格相隔方式或者大括号替代方式;

　 　　　　　2.均是通过事件冒泡方式，将事件传递到document进行事件的响应；

　　比较和联系:

　　　　　　1.bind()函数只能针对已经存在的元素进行事件的设置；但是live(),on(),delegate()均支持对未来新添加元素的事件设置；

　　　　　　2.bind()支持Jquery所有版本；live()支持jquery1.8-；delegate()支持jquery1.4.2+；on()支持jquery1.7+；

　　　　　　　bind()函数在jquery1.7版本以前比较受推崇，1.7版本出来之后，官方已经不推荐用bind()，替代函数为on(),这也是1.7版本新添加的函数，同样，可以用来代替live()函数，live()函数在1.9版本已经删除；

 　　　　　 3.live()函数和delegate()函数两者类似，但是live()函数在执行速度，灵活性和CSS选择器支持方面较delegate()差些；

**jQuery 的属性拷贝(extend)的实现原理是什么，何为深拷贝？何为浅拷贝？如何实现？**

extend()是jQuery中一个重要的函数，作用是实现对对象的扩展, 它经常用于jQuery插件的开发，jQuery内部也使用它来扩展属性方法。在jQuery的API手册中，我们看到，extend实际上是挂载在jQuery和jQuery.fn上的两个不同方法，尽管在jQuery内部jQuery.extend()和jQuery.fn.extend()是用相同的代码实现的，但是它们的功能却不太一样。

来看一下[官方API对extend](http://api.jquery.com/?s=extend)的解释:

jQuery.extend(): Merge the contents of two or more objects together into the first object.(把两个或者更多的对象合并到第一个当中)

jQuery.fn.extend():Merge the contents of an object onto the jQuery prototype to provide new jQuery instance methods.(把对象挂载到jQuery的prototype属性，来扩展一个新的jQuery实例方法)

我们知道，jQuery有静态方法和实例方法之分, 那么jQuery.extend()和jQuery.fn.extend()的第一个区别就是一个用来扩展静态方法，一个用来扩展实例方法。用法如下:

<**script**>  
 jQuery.extend({  
 sayhello:**function**(){  
 **console**.log(**"Hello,This is jQuery Library"**);  
 }  
 })  
 $.sayhello(); *//Hello, This is jQuery Library* jQuery.fn.extend({  
 check: **function**() {  
 **return this**.each(**function**() {  
 **this**.**checked** = **true**;  
 });  
 },  
 uncheck: **function**() {  
 **return this**.each(**function**() {  
 **this**.**checked** = **false**;  
 });  
 }  
 })  
 $( **"input[type='checkbox']"** ).check(); *//所有的checkbox都会被选择*</**script**>

注意两种调用插件的方式，一种是直接用$调用，另外一种是用$()调用，另外jQuery.extend()接收多个对象作为参数，如果只有一个参数，则把这个对象的属性方法附加到jQuery上，如果含有多个参数，则把后面的对象的属性和方法附加到第一个对象上。jQuery extend的实现源码:

<**script**>  
 jQuery.extend = jQuery.fn.extend = **function**() {  
 **var** options, name, src, copy, copyIsArray, clone,  
 target = arguments[0] || {},  
 i = 1,  
 length = arguments.**length**,  
 deep = **false**;  
 *// Handle a deep copy situation* **if** ( typeof target === **"boolean"** ) {  
 deep = target;  
 target = arguments[1] || {};  
 *// skip the boolean and the target* i = 2;  
 }  
 *// Handle case when target is a string or something (possible in deep copy)* **if** ( typeof target !== **"object"** && !jQuery.isFunction(target) ) {  
 target = {};  
 }  
 *// extend jQuery itself if only one argument is passed* **if** ( length === i ) {  
 target = **this**;  
 --i;  
 }  
 **for** ( ; i < length; i++ ) {  
 *// Only deal with non-null/undefined values* **if** ( (options = arguments[ i ]) != **null** ) {  
 *// Extend the base object* **for** ( name in options ) {  
 src = target[ name ];  
 copy = options[ name ];  
 *// Prevent never-ending loop* **if** ( target === copy ) {  
 **continue**;  
 }  
 *// Recurse if we're merging plain objects or arrays* **if** ( deep && copy && ( jQuery.isPlainObject(copy) || (copyIsArray = jQuery.isArray(copy)) ) ) {  
 **if** ( copyIsArray ) {  
 copyIsArray = **false**;  
 clone = src && jQuery.isArray(src) ? src : [];  
 } **else** {  
 clone = src && jQuery.isPlainObject(src) ? src : {};  
 }  
 *// Never move original objects, clone them* target[ name ] = jQuery.extend( deep, clone, copy );  
 *// Don't bring in undefined values* } **else if** ( copy !== **undefined** ) {  
 target[ name ] = copy;  
 }  
 }  
 }  
 }  
 *// Return the modified object* **return** target;  
 };  
</**script**>

很大一堆代码，乍看起来难以理解，其实代码的大部分都是用来实现jQuery.extend()中有多个参数时的对象合并，深度拷贝问题，如果去掉这些功能，让extend只有扩展静态和实例方法的功能，那么代码如下:

<**script**>  
 jQuery.extend = jQuery.fn.extend = **function**(obj){  
 *//obj是传递过来扩展到this上的对象* **var** target=**this**;  
 **for** (**var** name in obj){  
 *//name为对象属性  
 //copy为属性值* **copy**=obj[name];  
 *//防止循环调用* **if**(target === **copy**) **continue**;  
 *//防止附加未定义值* **if**(typeof **copy** === **'undefined'**) **continue**;  
 *//赋值* target[name]=**copy**;  
 }  
 **return** target;  
 }  
</**script**>

下面再来对extend方法进行注释解释:

<**script**>  
 jQuery.extend = jQuery.fn.extend = **function**() {  
 *// 定义默认参数和变量  
 // 对象分为扩展对象和被扩展的对象   
 //options 代表扩展的对象中的方法  
 //name 代表扩展对象的方法名  
 //i 为扩展对象参数起始值  
 //deep 默认为浅复制* **var** options, name, src, copy, copyIsArray, clone,  
 target = arguments[0] || {},  
 i = 1,  
 length = arguments.**length**,  
 deep = **false**;  
 *//当第一个参数为布尔类型是，次参数定义是否为深拷贝  
 //对接下来的参数进行处理* **if** ( typeof target === **"boolean"** ) {  
 deep = target;  
 target = arguments[1] || {};  
 *// 当定义是否深拷贝时，参数往后移动一位* i = 2;  
 }  
 *// 如果要扩展的不是对象或者函数，则定义要扩展的对象为空* **if** ( typeof target !== **"object"** && !jQuery.isFunction(target) ) {  
 target = {};  
 }  
 *// 当只含有一个参数时，被扩展的对象是jQuery或jQuery.fn* **if** ( length === i ) {  
 target = **this**;  
 --i;  
 }  
 *//对从i开始的多个参数进行遍历* **for** ( ; i < length; i++ ) {  
 *// 只处理有定义的值* **if** ( (options = arguments[ i ]) != **null** ) {  
 *// 展开扩展对象* **for** ( name in options ) {  
 src = target[ name ];  
 copy = options[ name ];  
 *// 防止循环引用* **if** ( target === copy ) {  
 **continue**;  
 }  
 *// 递归处理深拷贝* **if** ( deep && copy &&; ( jQuery.isPlainObject(copy) || (copyIsArray = jQuery.isArray(copy)) ) ) {  
 **if** ( copyIsArray ) {  
 copyIsArray = **false**;  
 clone = src && jQuery.isArray(src) ? src : [];  
 } **else** {  
 clone = src && jQuery.isPlainObject(src) ? src : {};  
 }  
 target[ name ] = jQuery.extend( deep, clone, copy );  
 *// 不处理未定义值* } **else if** ( copy !== **undefined** ) {  
 *//给target增加属性或方法* target[ name ] = copy;  
 }  
 }  
 }  
 }  
 *//返回* **return** target;  
 };  
</**script**>

那么，何为深拷贝，浅拷贝呢？

先看语法：

jQuery.extend( [deep ], target, object1 [, objectN ] )

描述：将两个或更多对象的内容合并到第一个对象。

深浅拷贝对应的参数就是[deep]，是可选的，为true或false。默认情况是false（浅拷贝），并且false是不能够显示的写出来的。如果想写，只能写true（深拷贝）~~

什么是深、浅拷贝呢？先讲定义，再举例子

在默认情况下，通过$.extend()合并操作不是递归的（浅拷贝）;如果第一个对象的属性本身是一个对象或数组，那么它将完全用第二个对象相同的key重写一个属性。这些值不会被合并。然而，如果将 true 作为该函数的第一个参数，那么会在对象上进行递归的合并（深拷贝）。

浅拷贝（false 默认）：如果第二个参数对象有的属性第一个参数对象也有，那么不会进行相同参数内部的比较，直接将第一个对象的相同参数覆盖。

深拷贝（true）：如果第二个参数对象有的属性第一个参数对象也有，还要继续在这个相同的参数向下一层找，比较相同参数的对象中是否还有不一样的属性，如果有，将其继承到第一个对象，如果没有，则覆盖。

举个栗子：

$.extend浅拷贝：

<**script**>  
 **var *obj1*** = {**name**:xx,**age**:18,**sex**:man};  
 **var *obj2*** = {**name**:cc,**age**:18};  
 $.extend(***obj1***,***obj2***);  
 ***obj1***----->{**name**:cc,**age**:18} *//被obj2覆盖*</**script**>

$.extend深拷贝：

<**script**>  
 **var *obj1*** = {**name**:xx,**age**:18,**sex**:man};  
 **var *obj2*** = {**name**:cc,**age**:18};  
 $.extend(**true**,***obj1***,***obj2***);  
 ***obj1***----->{**name**:cc,**age**:18，**sex**:man}  
</**script**>

[**jQuery.fn.extend与jQuery.extend到底区别在哪？**](http://blog.csdn.net/joyhen/article/details/38750935)**（转载某博主文章，做参考，辅助理解）**

其实说白了，从两个方法本身就能看出来端倪。我们先把[jQuery](http://lib.csdn.net/base/jquery)看成了一个类，这样好理解一些。**jQuery.extend()，是扩展的jQuery这个类。**

假设我们把jQuery这个类看成是人类，能吃饭能喝水能跑能跳，现在我们用jQuery.extend这个方法给这个类拓展一个能唱歌的技能。这样的话，不论是男人，女人，xx人.....等能继承这个技能（方法）了。

我们这样写：

jQuery.extend({  
 liu:  
 **function**(){  
 alert(**'liu'**);  
 }  
});

然后：$.liu();这样就能打印出来”liu“这个字符串

<!doctype **html**>  
<**html lang="zh-cmn-Hans-CN"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>Document</**title**>  
 <**script type="text/javascript" src="jquery-2.1.1.js"**></**script**>  
 <**script type="text/javascript"**>  
 ;(**function**($) {  
 $.extend({  
 liu: **function**(){  
 alert(**'liu'**);  
 }  
 });  
 })(jQuery);  
 </**script**>  
 <**script type="text/javascript"**>  
 $(**document**).ready(**function**() {  
 $.liu();  
 });  
  
 </**script**>  
</**head**>  
<**body**></**body**>  
</**html**>

这说明啥啊？这说明.liu()变成了jQuery这个类本身的方法（object）嘛。他现在能”唱歌“了。

我们再看看jQuery.fn.extend()这个方法。从字面理解嘛，这个拓展的是jQuery.fn的方法。jQuery.fn是啥玩意呢？源码里是这么写的：

jQuery.**fn** = jQuery.**prototype** = {  
 init: **function**(  
 selector, context ) {*//….　  
 //……* };

哦，原来jQuery.fn=jQuery.prototype，就是原型啊。那就一目了然了，jQuery.fn.extend拓展的是jQuery对象（原型的）的方法啊！对象是啥？就是类的实例化嘛，例如$(#abc)，这个玩意就是一个实例化的jQuery对象嘛。那就是说，jQuery.fn.extend拓展的方法，你得用在jQuery对象上面才行啊！他得是张三李四王五痳六这些实例化的对象才能用啊。说白了就是得这么用（假设xyz()是拓展的方法）：$('selector').xyz();你要是这么用$.xyz()；是会出错误滴。

例子源码如下：

<!doctype **html**>  
<**html lang="zh-cmn-Hans-CN"**>  
<**head**>  
 <**meta charset="UTF-8"**>  
 <**title**>Document</**title**>  
 <**script type="text/javascript" src="jquery-2.1.1.js"**></**script**>  
 <**script type="text/javascript"**>  
 ;(**function**($) {  
 $.fn.extend({  
 liu: **function**(){  
 alert(**'liu'**);  
 }  
 });  
 })(jQuery);  
 </**script**>  
 <**script type="text/javascript"**>  
 $(**document**).ready(**function**() {  
 $(**'div'**).liu();  
 });  
  
 </**script**>  
</**head**>  
<**body**>  
<**div**></**div**>  
</**body**>  
</**html**>

 和上边的区别区别一目了然吧？

其实吧，**jQuery.extend()这个方法，主要是用来拓展全局函数**，例如$.ajax()这种，要不就是拓展个选择器啦，例如$.fn.each()，当选择器用。大部分插件都是用jQuery.fn.extend()。