JQuery Widget

<https://www.cnblogs.com/dc10101/archive/2012/05/03/2481004.html>

[**编写jQueryUI插件(widget)**](https://www.cnblogs.com/dc10101/archive/2012/05/03/2481004.html)

使用jQueryUI的widget来写插件，相比于基本的jquery插件有一些好处：

\* 方便实现继承，代码重用

\* 默认是单例

\* widget已经给你实现好的一些常用方法，例如destroy

Widget 定义：

$.widget('命名空间.插件名', $.继承插件的命名空间.插件名,{ /\* options snip \*/ }); - 需要继承

$.widget('命名空间.插件名', { /\* options snip \*/ }); - 没有继承

## 基本知识：命名规范，public, private, this, this.element

* 命名空间： 插件的名称必须包含命名空间， 您只能创建一层深的命名空间，因此，custom.progressbar 是一 个有效的插件名称，而 very.custom.progressbar 不是一个有效的插件名称。命名空间只是JQuery Widget 内部分类管理时使用的。 实际插件使用中并不需要关心命名空间。
* 在widget里面的函数，就有public和private之分，规则是：

public方法首字符不是 \_ （下划线）

private方法首字符是 \_ （下划线）

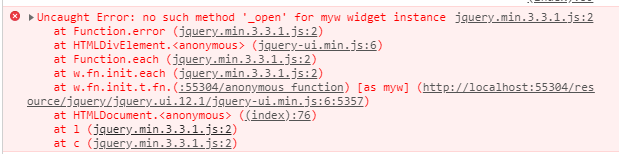
当调用方法时，会先判断是否以\_开头，如果是则不执行调用。

$.widget("wm.myw", {

\_open: function (a) {

console.log("\_Open: " + a);

}

});

$(function () {

$("#d1").myw();

$("#d1").myw("\_open", 100);

})

---------------------------------------------------------------------------------------------

$.widget("wm.myw", {

open: function (a, b, c) {

alert("open : " + a + ":" + b + ":" + c);

console.log("\_Open: " + a + ":" + b + ":" + c);

}

});

$(function () {

 $("#d1").myw();

$("#d1").myw("open", 100, 200, 300);

})

----------------------------------------------------------------------------------------------------------------------------------------------------

$(function () {

$("#d1").myw();

$("#d1").myw("open", [100, 200, 300]);

})

Widget对public function的调用，是执行 function.call(obj, arg1,arg2,...) 而不是执行： function.apply(obj, [arg1, arg2,…])

-------------------------------------------------------------------------------------------------------------

对于 Private 函数的使用：

$.widget("wm.myw", {

\_open: function (a, b) {

alert("private \_open: " + a + ":" + b);

console.log("private open: " + a + ":" + b);

},

open: function (a, b, c) {

this.\_open(a, b); - 可以在内部使用 this.\_xxxx() 来调用私有函数

alert("open : " + a + ":" + b + ":" + c);

console.log("open: " + a + " : " + b + " : " + c);

}

});

$(function () {

$("#d1").myw();

$("#d1").myw("open", 100, 200, 300);

});

如果我非要在外面调用private方法，该怎么做？并非一点办法也没有：

1. var instance = $('<div>');
2. instance.mywidget('publicFunction'); // work
3. instance.mywidget('\_privateFunction'); // silently fail
4. instance.data('mywidget').\_privateFunction(); // work – 本人测试不可以
5. $.mynamespace.mywidget.prototype.\_privateFunction(); // work

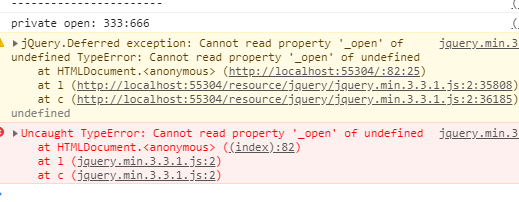
$(function () {

$("#d1").myw();

$.wm.myw.prototype.\_open(333, 666); - 正常工作

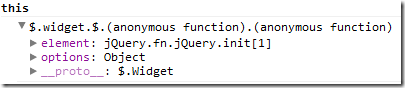
$("#d1").data("myw").\_open(888, 999); - 不工作

});



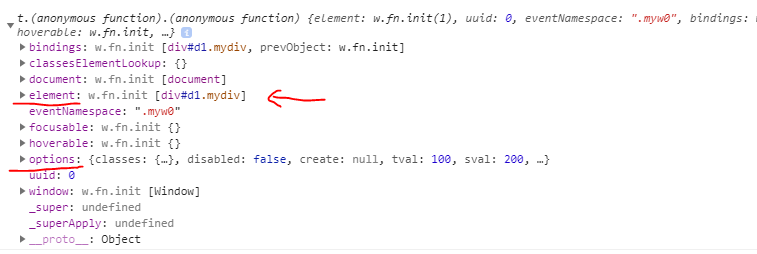
一般来说工具函数写在widget外面比较合适，但如果你想要这些工具函数被子类继承，则需要写在widget里面

* 在widget内，this表示的是什么？我们在widget的一个public函数内用console.log(this)打出来瞧瞧：

[](http://images.cnblogs.com/cnblogs_com/dc10101/201205/201205031803317892.png)

日志显示，this是一个$.widget.$.(anonymous function).(anonymous function)

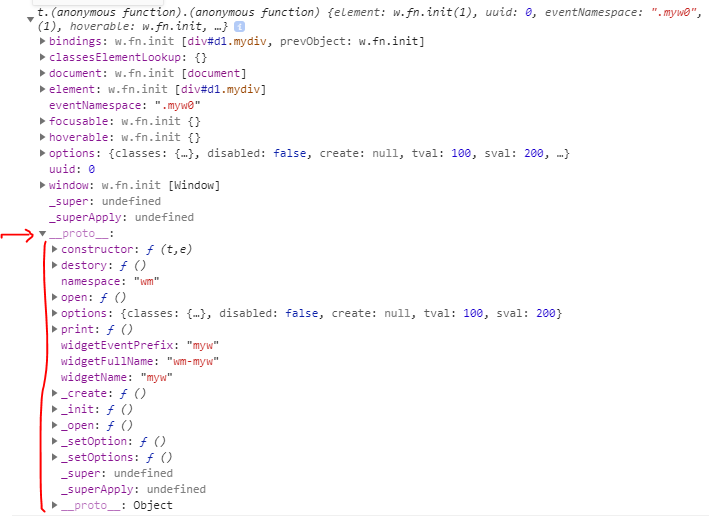
this.element是变成widget的那个jQuery对象，如果要用jquery的方法，往往首先要取到jquery对象。



this.element 直接就是JQuery 对象， 不需要使用 $(this.element).addClass() , 直接使用 this.element.addClass()

this.\_\_proto\_\_包含了插件中定义的所有public和private函数，以及继承过来的方法。

这里简单介绍一下\_\_proto\_\_：每个对象都会在其内部初始化一个属性，就是\_\_proto\_\_，当我们访问一个对象的属性 时，如果这个对象内部不存在这个属性，那么他就会去\_\_proto\_\_里找这个属性，这个\_\_proto\_\_又会有自己的\_\_proto\_\_，于是就这样 一直找下去，也就是我们平时所说的原型链的概念。



## \_create  \_init    destroy

widget factory实现了一种单例模式，即不允许在同一个jQuery对象上多次实例化。

当调用$(XX).widgetName()进行初始化的时候，会执行以下代码（源码截取自jquery.ui.widget.js）：

var instance = $.data( this, name ); // 从widget自身取出名字为name的数据

if ( instance ) {

instance.option( options || {} ).\_init(); // 若该数据已经存在则只调用\_init

} else {

$.data( this, name, new object( options, this ) ); // 若数据还没有则新建一个实例，并将实例保存

}

当调用$(XX).widgetName(‘destroy’)进行销毁的时候，执行以下代码（源码截取自jquery.ui.widget.js）：

this.element

.unbind( "." + this.widgetName )

.removeData( this.widgetName ); // 删除在create时保存的数据

有一个removeData的操作，那么下次调用$(XX).widgetName()就会重新实例化了。

需要注意的是，destroy方法在jquery.ui.widget.js中是有默认实现的，而\_create和\_init没有实现。因此如果用自己的方法覆盖destroy，不要忘记调用默认的：

destory: function () {

console.log('destory');

// call the original destroy method since we overwrote it

$.Widget.prototype.destroy.call(this);

}

以下示例代码验证\_create和\_init的区别以及destroy的作用：

var mw = $('#test').myWidget(); // \_create \_init

mw = $('#test').myWidget(); // \_init

mw.myWidget('destory');

mw = $('#test').myWidget(); // \_create \_init

$.widget("wm.myw", {

options: {

tval: 100,

sval: 200

},

\_init: function () {

console.log("-----------------------");

console.log("\_init");

console.log(this.options);

this.element.append(" Init Hello JQuery");

},

\_create: function () {

console.log("-----------------------");

console.log("\_create");

this.element.append(" Create Hello JQuery");

},

\_setOptions: function () {

console.log("-----------------------");

console.log("\_setOptions");

console.log(this.options);

this.\_superApply(arguments);

},

\_setOption: function (key, val) {

console.log("\_setOption: " + key + "=" + val);

this.\_super(key, val);

},

destory: function () {

console.log("destory");

$.Widget.prototype.destroy.call(this);

},

print: function (a, b, c) {

//this.\_trigger("showMe", event, 100, 200);

console.log(`print: ${a} = ${b} = ${c}`);

console.log(this.options);

}

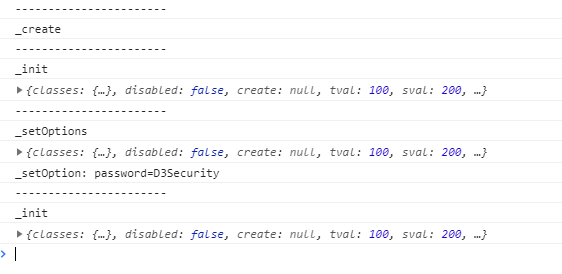
});

$(function () {

$("#d1").myw({user:"william"}); - 执行 Create, Init , 并不执行\_setOptions

$("#d1").myw({password: "D3Security"}); - 执行 \_setOptions, \_setOption, Init

});



# 那么在\_create和\_init以及destroy里分别应该做什么：

\_create: 生成HTML，事件绑定。

\_init: 执行默认的初始化动作，例如把页面变成初始状态。

destory: 调用$.Widget.prototype.destroy.call(this)，删除HTML。

注意：绑定事件要注意给事件名加命名空间后缀：例如 .bind('mouseenter.mywidget', this.\_hover)

## options

widget默认实现了两个函数：\_setOptions 和\_setOption，

\_setOptions 的实现就是对每个要修改的 option调用\_setOption，也就是说真正修改的动作在\_setOption里。

因此，如果要重写\_setOption函数，则一定不要忘记写：

$.Widget.prototype.\_setOption.apply(this, arguments);

或者 this.options[key] = val; 也可以

或者 this.\_super(key, val); -- 最新的写法, 相当于 js function.call

\_setOptions: function () {

console.log("-----------------------");

console.log("\_setOptions");

console.log(arguments);

this.\_superApply(arguments); -- 这个很重要，如果没有的话，将不会往下继续触发 \_setOption 函数，数据

也就无法增删改

},

\_setOption: function (key, val) {

console.log("\_setOption: " + key + "=" + val);

this.\_super(key, val);

//this.options[key] = val;

//$.Widget.prototype.\_setOption.apply(this, arguments);

},

调用\_setOption 之前，一定是先调用 \_setOptions 来触发 \_setOption

# 何时会调用\_setOption的结论：

1. 像instance.mywidget('option', 'field1', 2); 这样显式设置option时。

$("#d1").myw("option", "url", "http://www.yahoo.com");

- 会先调用 \_setOptions , 再调用 \_setOption, 并不调用 \_init

2. 带着options初始化时：

如果实例不存在，即需要调用\_create，则不调用\_setOption；

如果实例已存在，仅需要调用\_init，则会在调用\_init之前调用\_setOption。

$("#d1").myw({user:"william"}); - 实例不存在，调用 \_create, \_init, 不调用 \_setOptions, \_setOption

$("#d1").myw({ password: "D3Security", hashcode: "13453" }); - 实例存在, 调用\_setOptions, \_setOption, 再调用 \_init

3. 内部函数使用 this.options.xxx = value 来修改值时， 并不触发任何事件包括 \_setOption

change: function (a) {

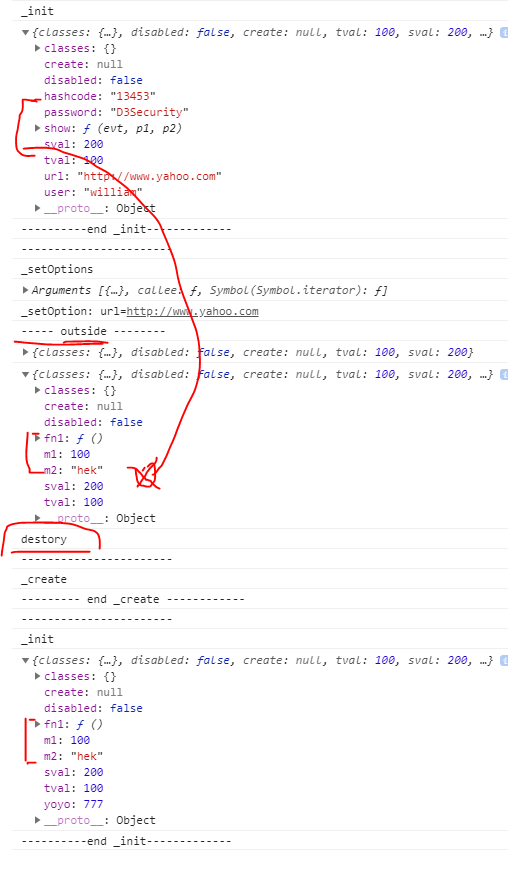
this.options.url = a;

}

$("#d1").myw("change", "www.sohu.com");

4. 至于何时需要定义：\_setOptions, \_setOption. 如果 options 的属性值改变时，需要添加业务逻辑。否则可以不定义这

两个内部方法。使用默认的定义即可。

5. 如何从Widget 外部调用 options:

实例的options克隆自widget.protottype.options

$.mynamespace.mywidget.prototype.options

$.wm.myw.prototype.options - 修改原型的options, 并不会影响已经存在的实例options，因为他们是克隆的副本。 只会对以后创建的实例有影响。

$(function () {

$("#d1").myw({

user: "william",

show: function (evt, p1, p2) {

console.log(evt);

console.log(`parameters: ${p1} => ${p2}`);

alert(`parameters: ${p1} => ${p2}`);

}

});

$("#d1").myw({ password: "D3Security",

hashcode: "13453"

});

$("#d1").myw("option", "url", "http://www.yahoo.com");

console.log("----- outside --------");

console.log($.wm.myw.prototype.options);

$.extend($.wm.myw.prototype.options,

{

m1: 100,

m2: "hek",

fn1: function () { alert("OKey"); }

});

console.log($.wm.myw.prototype.options);

$("#d1").myw("destory");

$("#d1").myw({ yoyo:777});

});

## \_trigger

注意这个\_trigger是jQueryUI widget factory里的，和jQuery里$.fn命名空间下的trigger函数不是一个东西（后者不带下划线）。

\_trigger一般用来回调用户传入options的callback。

在插件内部调用\_trigger(‘myEvent’)即相当于调用options里面的myEvent这个回调函数。

要改动options里的event handler应该怎么做呢？不要使用bind/unbind，而是去修改options：

// bind (overwrite, not add event handler)

mw.myWidget('option', 'myEvent', function (event, ui) {

console.log('new implement');

});

// unbind

mw.myWidget('option', 'myEvent', null);

注意： \_trigger 只是用于内部触发 options 里定义的函数，并不用来触发widget的public 或private 函数

$.widget("wm.myw", {

showp: function (a) {

this.\_trigger("**show**", null, a); - 属于function.apply(this, [arg1, arg2,…])

}

});

$(function () {

$("#d1").myw({

user: "william",

**show**: function (evt, p1, p2) {

console.log(evt); -- 上面传递的是 null 值

console.log(`parameters: ${p1} => ${p2}`);

alert(`parameters: ${p1} => ${p2}`);

}

});

$("#d1").myw("showp", [200, 300]); - 参数必须放在数组里

});

注意：

1. options 里的函数， 第一个参数是固定的，是用于接收事件 event 参数 ，即使我们调用时传递null值， 最终还是会自动赋值 event

**show**: function(evt, p1, p2) { console.log(evt); … }



1. 由于是function.apply(obj, [arg1, arg2,…]), 所以给函数传递参数必须使用数组：

**show**: function (evt, p1, p2) {…}

showp: function (a) {

this.\_trigger("**show**", null, a);

}

$("#d1").myw("showp", [200, 300]); - 参数必须放在数组里

1. 如果使用this.\_trigger 来触发widget 内部的public, private 函数， 则不执行任何动作，也不报错

# [jQuery编写widget的一些窍门](https://www.cnblogs.com/dail/archive/2010/10/27/1862597.html)

1. 在编写widget的时候，一般我们需要绑定一些事件，最好将这些widget的绑定事件加上当前widget的命名空间。如果同一个jQuery对象，使用了两个widget，而两个widget都绑定了相同的事件名称，可能会出现问题。在销毁widget的时候，去除绑定事件也很方便，只需要unbind（”.namespace“）就可以了。
2. 在写jQuery的时候，因为jQuery对象是支持连写的。譬如：$(obj).css("height","20px").attr("title","abc")....
3. 在编写的时候可以利用javascript的原生方法来避免switch。

[复制代码](javascript:void(0);)

switch(a)  
{  
　　case "aa":  
　　　　this.\_set\_aa();  
　　　　break;  
　　case "bb":  
　　　　this.\_set\_bb();  
　　　　break;  
　　case "cc"  
　　　　this.\_set\_cc();  
　　　　break;  
}

[复制代码](javascript:void(0);)

上面的代码可以用下面的代码替换

this["\_set\_"+a]();

1. 尽量缓存jQuery对象，和各种变量。这样可以提高脚本的性能
2. 采用变量缓存this指针，在最小化代码的时候this可以被最小化掉。
3. 最好将css类名取一致的名称，然后定义变量保存起来。在使用的时候直接采用变量，这样就算css名称有调整，只需要改变变量缓存的值即可。同时代码在最小化的时候也能减小体积。
4. setOption的时候，如果一个option是复杂对象，而不是简单的值对象，最好不要简单的this.options[key]=value.而在这之前需要将value和以前的option的值做一个extend，然后在赋值，这样可以保留复杂对象原有的部分值。比如：

var a = {width:120,height:200};  
var c={width:200};  
a=c;  
a=$.extend(a,c);

 其结果是第一个a将是{width:20},会丢失掉height：200；而下面的那个将继续保留原始的height:200。

需要特别注意的是： destroy 会重置 options, 但是并不会释放自己绑定的事件，必须自己解除绑定。否则会有多次绑定同一个事件。造成重复执行多次同样事件的代码逻辑错误。

$.widget("wliu.tab9", {

options: { tabsn: 0 },

\_create: function () {

$("li", this.element).not(":has(span)").each(function (idx, el) {

var text = $(el).text();

$(el).empty();

$(el).append("<span>" + text + "</span>");

});

$("li", this.element).not(":has(s)").append("<s>");

// 先解除事件绑定 off，避免下次重复绑定. on

var self = this;

$("li", this.element).off("click.tab9").on("click.tab9", function (evt) {

self.options.tabsn++; // 这里面使用 this 是很混淆的，this 并不是widget对象

console.log(self.options.tabsn);

});

console.log("tab9 created:" + this.element.attr("id"));

},

\_init: function () {

console.log("tab9 init: " + this.element.attr("id"));

},

destroy: function () {

console.log("destroy");

$("li", this.element).off("click.tab9"); - 解除事件绑定，避免下次重复绑定

$.Widget.prototype.destroy.call(this);

}

});

$.widget("wliu.tab9", {

options: {

tabsn: 0,

height: 0,

color: ""

},

\_create: function () {

$("li", this.element).not(":has(span)").each(function (idx, el) {

var text = $(el).text();

$(el).empty();

$(el).append("<span>" + text + "</span>");

});

$("li", this.element).not(":has(s)").append("<s>");

$("li", this.element).each(function (idx, el) {

$(el).attr("tabsn", idx);

});

$("+div[wliu][tab9][body]>div", this.element).each(function (idx, el) {

$(el).attr("tabsn", idx);

});

var self = this;

$("li", this.element).off("click.tab9").on("click.tab9", function (evt) {

self.\_setOption("tabsn", $(this).attr("tabsn"));

});

this.\_changeHeight();

this.\_changeColor();

this.\_changeTab();

},

\_init: function () {

},

\_setOption: function (key, val) {

this.\_super(key, val);

switch (key) {

case "height":

this.\_changeHeight();

break;

case "tabsn":

this.\_changeTab();

break;

case "color":

this.\_changeColor();

break;

}

},

\_changeTab: function () {

$("li", this.element).attr("selected", null);

$("+div[wliu][tab9][body]>div", this.element).attr("selected", null);

if ($(`li[tabsn="${this.options.tabsn}"]`, this.element).length <= 0) {

this.options.tabsn = 0;

}

$(`li[tabsn="${this.options.tabsn}"]`, this.element).attr("selected", "");

$(`+div[wliu][tab9][body]>div[tabsn="${this.options.tabsn}"]`, this.element).attr("selected", "");

},

\_changeHeight: function () {

if (this.options.height > 0)

$("+div[wliu][tab9][body]", this.element).height(this.options.height);

else

$("+div[wliu][tab9][body]", this.element).height("auto");

},

\_changeColor: function () {

if (("" + this.options.color).trim() === "")

this.element.attr("color", null);

else

this.element.attr("color", this.options.color);

},

destroy: function () {

$("li", this.element).off("click.tab9");

$.Widget.prototype.destroy.call(this);

}

});