# 原生js控件编写说明

## jiraBox控件的完整示例代码

;(**function**(){  
 **var** defaults = {  
 *//各种参数，方法* **originalData**: [],  
 **data**: [],  
 **allData**: [],  
 **url**: **""**,  
  
 init: **function**(elem){},  
 autoSubmit: **function**(elem){},  
 submit: **function**(elem){},  
 cancel: **function**(elem){},  
 edit: **function**(elem){},  
 showList: **function**(elem){},  
 restore: **function**(elem){},  
 scrollDownHandler: **function**(elem){},  
 scrollUpHandler: **function**(elem){},  
  
 **render**: {  
 renderJiraBox: **function**(elem){},  
 renderEdit: **function**(elem){},  
 reRenderCursor: **function**(elem){},  
 reRenderBoxUl: **function**(elem){},  
 reRenderSelectUl: **function**(elem){}  
 },  
  
 **eventListener**: {  
 aClick: **function**(e){},  
 bindEditEvent: **function**(e){},  
 submitByEvent: **function**(e){},  
 cancelByEvent: **function**(e){},  
 showListByEvent: **function**(e){},  
 selectByEvent: **function**(e){},  
 deleteSelectedByEvent: **function**(e){},  
 inputPropertyChangeListener: **function**(e){},  
 keydownListener: **function**(e){},  
 windowClickHandler: **function**(e){},  
 resizeHandler: **function**(){}  
 }  
 };  
  
 **var** api ={  
 init: **function** *listen*(elem, opts){  
 *//这里通过typeof设置监听的元素需为字符串调用，实际可根据需要进行更改* **if** (**typeof** elem === **'string'**) {  
 **var** elems = **document**.querySelectorAll(elem);  
 **var** i = elems.**length**;  
 *//通过递归将listen方法应用在所有的dom元素上* **while** (i--) {  
 **this**.init(elems[i], opts); *//typeof dom 是 object* }  
 **return this**;  
 }  
  
 *//在这里，你可以将插件的部分功能函数写在这里* **if**(elem.**jiraBox**){  
 Object.assign(elem.**jiraBox**.**options**, opts);  
 }**else**{  
 elem.**jiraBox** = {  
 **options**: Object.assign({}, defaults, opts),  
 **api**: api  
 };  
 }  
  
 defaults.init(elem);  
 **return this**;  
 },  
 options: **function**(elem){},  
 getValue: **function**(elem){},  
 setValue: **function**(elem, data){},  
 loadData: **function**(elem, data){},  
 edit: **function**(elem){},  
 submit: **function**(elem){},  
 cancel: **function**(elem){},  
 enable: **function**(elem){},  
 disable: **function**(elem){},  
 showError: **function**(elem, msg){},  
 hideError: **function**(elem){},  
 showLoading: **function**(elem){},  
 hideLoading: **function**(elem){}  
 };  
  
 **this**.**jiraBox** = api; *//只把api暴露出去*})();

以上就是原生js开发组件的基本模板。

## 详细步骤说明

### 1、准备架子

;(**function**(){  
 *//插件所以功能写在里面*})();

说明：因为是原生js，所以不需要传参数，写一个立即执行函数就可以了。

### 2、定义控件默认的参数和方法

**var** defaults = {  
 *//各种参数，方法*  
 **param**: [],  
 ...  
 ...  
 func: **function**(){...},  
 };

说明：只是定义，会在组件内其他的地方使用。参数和方法都应该只是组件使用，不应该由用户直接操作它们。例如，用户想修改某个data属性，而且用户知道数据都存放在pluginName.options中，就通过赋值来修改，这样是能达到效果，但不应该这样处理，而是通过暴露的方法如pluginName.setData(..)来修改。

### 3、控件供用户使用的函数

**var** api ={  
 func: **function**(..){... },

...  
 };

**this**.jiraBox = api; *//只把api暴露出去*

说明：在api对象中定义一系列的方法，然后通过this.pluginName = api; 我们就能在js中通过pluginName.func(..)来调用方法。

### 4、初始化及链式操作

（1）我们以示例代码的api.init函数作为说明：

init: **function**(elem, opts){  
  
 *//这里通过typeof设置监听的元素需为字符串调用，实际可根据需要进行更改* **if** (**typeof** elem === **'string'**) {  
 *//这里使用ES5的querySelectorAll方法获取dom元素* **var** elems = **document**.querySelectorAll(elem),  
 i = elems.**length**;  
 *//通过递归将应用在所有的dom元素上* **while** (i--) {  
 **this**.init(elems[i], opts); *//typeof dom 是 object* }  
 **return this**;  
 }  
  
 *//在这里，你可以将插件的部分功能函数写在这里* **if**(elem.**jiraBox**){  
 Object.assign(elem.**jiraBox**.**options**, opts);  
 }**else**{  
 elem.**jiraBox** = {  
 **options**: Object.assign({}, defaults, opts),  
 **api**: api  
 };  
 }  
 defaults.init(elem); *//不是这里的api.init* **return this**;  
}

以上是通过jiraBox.init("#a", {...})来调用。第一个参数是选择器的字符串，但如果只能是id选择器就会写很多的重复代码，所以也可以接受其它所有css能识别的选择器，然后在函数中通过递归（遍历也可以）对选择的每个dom节点都执行相同的初始化，dom节点也是object对象，可以为它加上jiraBox属性，把有用的数据都存进该属性中，达到元素与数据绑定，最后return this返回控件对象就能继续.xxx()操作。

注意：这里的链式操作是对返回控件对象的方法进行链式操作，而不是dom元素的链式操作，即不能jiraBox.xxx().innerHTML = ""; 这样是会报错的。

（2）从下面代码看到，初始化的时候先判断是否具有pluginName属性，有就直接合并新的options，没有就赋pluginName属性。

**if**(elem.**pluginName**){  
 Object.assign(elem.**pluginName**.**options**, opts);  
}**else**{  
 elem.**pluginName** = {  
 **options**: Object.assign({}, defaults, opts),  
 **api**: api  
 };  
}

例如我们会这样初始化：

pluginName.init("#a", {

data: [],

field: "xx",

onSuccess: function(){},

onSelect: function(){}

})

通过Object.assign把用户提供的参数和方法与控件默认的参数和方法进行合并，这样控件就能执行用户自定义的函数。

在控件内部可以这样来处理：

*//确定按钮事件*element.querySelector(**".xxx"**).addEventListener(**"click"**,state.**options**.onSelect);

然后如果用户在options中提供onSelect，在合并之后，这个监听事件就是执行用户提供的自定义事件。

注意：这样合并是只要用户有提供，都会合并，可能会产生某个方法并不希望用户改变但被用户修改了导致控件无法正常运行。如某个控件文档，并没有提供finder属性，但在defaults中有一个finder的属性，如果用户不遵循文档或者拼写错误，pluginName.init({finder: "xx"})把属性修改了，就会导致控件报错并无法正常运行。

解决：可以把defaults分成两个，一个是一一对应文档的，用户可以自定义替换，即直接合并；另一个是不被替换的属性和方法对象。

（3）**关于合并的说明**：例如出现这样的一个现象，用户在options中设置onLoad，然后组件默认的onLoad事件是具有一系列的样式效果如显示加载中，这样如果用户自定义的函数直接替换了原来的onLoad，就会导致用户不知道怎么处理相关的样式，所以在设计上需要根据实际情况进行更改。下面是easyui的示例：

//先在defaults中定义空函数

onBeforeLoad: **function**(param){},  
 onLoadSuccess: **function**(data){},

...

easyui把提供自定义处理的函数都设置为空，不把控件的默认处理写在里面，然后在某个函数中根据逻辑进行调用：

*/\*\*  
 \* request remote data if the url property is setted.  
 \*/* **function** *request*(target, url, param, remainText){  
 **var** opts = $.data(target, **'combobox'**).**options**;  
 **if** (url){  
 opts.**url** = url;  
 }  
 param = $.extend({}, opts.**queryParams**, param||{});  
*// param = param || {};* **if** (opts.onBeforeLoad.call(target, param) == **false**) **return**;  
  
 opts.loader.call(target, param, **function**(data){  
 *loadData*(target, data, remainText);  
 }, **function**(){  
 opts.onLoadError.apply(**this**, arguments);  
 });  
 }

可以理解为不把request作为options中的属性，而是把request方法中用到的onBeforeLoad、onLoadError等作为options中的属性，这样保留控件原有的处理并执行默认的空方法或用户替换后的方法，用户可以在自定义函数中通过返回false来阻止控件的默认行为。

根据easyui的例子，我们可以把之前提到的

element.querySelector(**".xxx"**).addEventListener(**"click"**,state.**options**.onSelect);

进行修改:

element.querySelector(**".xxx"**).addEventListener(**"click"**, xxxFunc); //不调用options中的方法

function xxxFunc(){

.....

var state = xx.pluginName;

if(state.options.onSelect() == false){return;}

.....

}

easyui也有直接替换的情况，但它似乎是做了相应的分类，看下图：



loader、loadFilter是属于属性一栏，onBeforeLoad是属于事件一栏，都是函数类型，属性类的用户可替换，但替换后需要自己进行处理，事件类的是替换空函数。我们看它的源码对比：

....

loader: **function**(param, success, error){  
 **var** opts = $(**this**).combobox(**'options'**);  
 **if** (!opts.**url**) **return false**;  
 $.ajax({  
 **type**: opts.**method**,  
 **url**: opts.**url**,  
 **data**: param,  
 **dataType**: **'json'**,  
 success: **function**(data){  
 success(data);  
 },  
 error: **function**(){  
 error.**apply**(**this**, arguments);  
 }  
 });  
 },

.....

loadFilter: **function**(data){  
 **return** data;  
 },

....

onBeforeLoad: **function**(param){},

....

最后，我们回看一下上面提到的easyui的request函数，里面有一个opts.loader.call，即用户如果提供了自己的loader属性，函数体为空，就已经破坏了正常的request行为，需要用户自行实现，而事件类的原本就为空，就算用户提供了该函数，对request正常逻辑没有影响，是看作额外的处理。综上所述，用户提供的参数中函数类型的需要根据情况，是作为直接替换还是额外的处理，对于替换，就直接函数体替换，对于额外的处理，那原来的函数体就应该为空。