(**function** (factory) {

window["Sortable"] = factory();

})(**function** () {

Sortable.create = **function** (el, options) {  
 **return new** Sortable(el, options);  
};  
  
  
// Export  
Sortable.version = '1.4.2';  
**return** Sortable;

});

浏览器版本

var userAgent = navigator.userAgent.toLowerCase();   
browser={   
version: (userAgent.match( /.+(?:rv|it|ra|ie)[\/: ]([\d.]+)/ ) || [0,'0'])[1],   
safari: /webkit/.test( userAgent ),   
opera: /opera/.test( userAgent ),   
msie: /msie/.test( userAgent ) && !/opera/.test( userAgent ),   
mozilla: /mozilla/.test( userAgent ) && !/(compatible|webkit)/.test( userAgent )   
}

**var *\_navig*** = **function** () {  
 **var** Sys = {};  
 **var** ua = navigator.userAgent.toLowerCase();  
 **var** s;  
 (s = ua.match(/msie ([\d.]+)/)) ? Sys.ie = s[1] :  
 (s = ua.match(/firefox\/([\d.]+)/)) ? Sys.firefox = s[1] :  
 (s = ua.match(/chrome\/([\d.]+)/)) ? Sys.chrome = s[1] :  
 (s = ua.match(/opera.([\d.]+)/)) ? Sys.opera = s[1] :  
 (s = ua.match(/version\/([\d.]+).\*safari/)) ? Sys.safari = s[1] : 0;  
 **return** Sys;  
}();

浏览器版本号

<!--[if IE]>   
<script type="text/javascript">   
alert("ie");   
</script>   
<![endif]-->   
<!--[if IE 6]>   
<script type="text/javascript">   
alert("ie6");   
</script>   
<![endif]-->   
<!--[if IE 7]>   
<script type="text/javascript">   
alert("ie7");   
</script>   
<![endif]-->

创建事件 analog

if (document.createEvent) {

                    event = document.createEvent('HTMLEvents');

                    event.initEvent('click');

                    event.eventType = 'message';

                    el1.dispatchEvent(event);

                };

// for IE8-

if (document.createEventObject) {

                    event = document.createEventObject();

                    event.eventType = 'message';

                    el1.fireEvent('onclick', event);

 };

e.pageX

e.pageY

事件发生的位置

$().position().left

$().position().right

元素的绝对位置

$().offset().left

$().offset ().right

元素的相对位置

阻止冒泡

e.stopPropagation();

阻止默认事件

e.preventDefault()

事件冒泡（的过程）：事件从发生的目标（event.srcElement||event.target）开始，沿着文档逐层向上冒泡，到document为止。

**js事件委托**

js事件委托，其实是使用了冒泡的原理，从点击的元素开始，递归方式的向父元素传播事件，这样做的好处是对于大量要处理的元素，不必为每个元素都绑定事件，只需要在他们的父元素上绑定一次即可，提高性能。 还有一个好处就是可以处理动态插入dom中的元素，直接绑定的方式是不行的。

就是事件目标自身不处理事件，而是把处理任务委托给其父元素或者祖先元素，甚至根元素事件委托很好地利用了"事件冒泡"。当点击子元素，根据"事件冒泡"，该子元素的父级元素捕获了该次点击事件，并触发自己的方法。

由于事件委托是通过事件冒泡实现的，所以如果子级的元素阻止了事件冒泡，那么事件委托也将失效！

# $("#info\_table td").live("click",function(){/\*显示更多信息\*/});

这里的.live()方法会把click事件绑定到$(document)对象（但这一点从代码中体现不出来，这也是.live()方法饱受诟病的一个重要原因，稍后再详细讨论），而且只需要给$(document)绑定一次（不是50次，更不是5000次）。在接收到任何事件时，$(document)对象都会检查事件类型和事件目标，如果是click事件且事件目标是td，那么就执行委托给它的处理程序。

一切似乎很完美。可惜，事实并非如此。因为.live()方法并不完美，它有如下几个主要缺点：

$()函数会找到当前页面中的所有td元素并创建jQuery对象，但在确认事件目标时却不用这个td元素集合，而是使用选择符表达式与event.target或其祖先元素进行比较，因而生成这个jQuery对象会造成不必要的开销；

默认把事件绑定到$(document)元素，如果DOM嵌套结构很深，事件冒泡通过大量祖先元素会导致性能损失；

只能放在直接选择的元素后面，不能在连缀的DOM遍历方法后面使用，即$("#infotable td").live...可以，但$("#infotable").find("td").live...不行；

收集td元素并创建jQuery对象，但实际操作的却是$(document)对象，令人费解。

而为了避免事件冒泡造成的性能损失，jQuery从1.4开始支持在使用.live()方法时配合使用一个上下文参数：

$("td",$("#info\_table")[0]).live("click",function(){/\*显示更多信息\*/});

这样，“受托方”就从默认的$(document)变成了$("#infotable")[0]，节省了冒泡的旅程。

# 而为了解决无谓生成元素集合的问题，jQuery 1.4.2干脆直接引入了一个新方法.delegate()。

# $("#info\_table").delegate("td","click",function(){/\*显示更多信息\*/});

jQuery 1.7为了解决.**bind**()、.live()和.delegate()并存造成的不一致性问题，将会增加一对新的事件方法：.on()和.off()：

$(elems).on(events, selector, data, fn);

$(elems).off(events, selector, fn);

如果指定selector，则为事件委托；否则，就是常规绑定。

所以现在只要用on方法就可以了，推荐用on方法进行委托或常规绑定事件

ie 缺少的方法

**if**(!("console" in window)){  
 window.console={  
 log:**function** () {  
 },  
 error:**function** () {  
 },  
 warn:**function** () {  
 }  
 }  
}  
**if**(!("trim" in String.prototype)){  
 String.prototype.trim=**function** () {  
 **return this**.replace(/^\s\*/,"").replace(/\s\*$/,"");  
 }  
}  
**if**(!("remove" in Array.prototype)){  
 Array.prototype.remove = **function**(val) {  
 **var** index = **this**.indexOf(val);  
 **if** (index > -1) {  
 **this**.splice(index, 1);  
 }  
 };  
}  
**if**(!("indexOf" in Array.prototype)){  
 Array.prototype.indexOf = **function**(elt */\*, from\*/*)  
 {  
 **var** len = **this**.length >>> 0; **var** from = Number(arguments[1]) || 0;  
 from = (from < 0)  
 ? Math.ceil(from)  
 : Math.floor(from);  
 **if** (from < 0)  
 from += len;  
 **for** (; from < len; from++)  
 {  
 **if** (from in **this** &&  
 **this**[from] === elt)  
 **return** from;  
 }  
 **return** -1;  
 };  
}

//ie8 splice(int) bug  
// check if it is IE and it's version is 8 or older  
**if** (document.documentMode && document.documentMode < 9) {  
 // save original function of splice  
 **var *originalSplice*** = Array.prototype.splice;  
  
 // provide a new implementation  
 Array.prototype.splice = **function**() {  
 **var** arr = [], i = 0, max = arguments.length;  
  
 **for** (; i < max; i++){  
 arr.push(arguments[i]);  
 }  
  
 // if this function had only one argument  
 // compute 'deleteCount' and push it into arr  
 **if** (arr.length==1) {  
 arr.push(**this**.length - arr[0]);  
 }  
  
 // invoke original splice() with our new arguments array  
 **return *originalSplice***.apply(**this**, arr);  
 };  
}

## 浏览器加载

### 传统方法

在 HTML 网页中，浏览器通过<script>标签加载 JavaScript 脚本。

<!-- 页面内嵌的脚本 -->

<script type="application/javascript">

// module code

</script>

<!-- 外部脚本 -->

<script type="application/javascript" src="path/to/myModule.js">

</script>

上面代码中，由于浏览器脚本的默认语言是 JavaScript，因此type="application/javascript"可以省略。

默认情况下，浏览器是同步加载 JavaScript 脚本，即渲染引擎遇到<script>标签就会停下来，等到执行完脚本，再继续向下渲染。如果是外部脚本，还必须加入脚本下载的时间。

如果脚本体积很大，下载和执行的时间就会很长，因此成浏览器堵塞，用户会感觉到浏览器“卡死”了，没有任何响应。这显然是很不好的体验，所以浏览器允许脚本异步加载，下面就是两种异步加载的语法。

<script src="path/to/myModule.js" defer></script>

<script src="path/to/myModule.js" async></script>

上面代码中，<script>标签打开defer或async属性，脚本就会异步加载。渲染引擎遇到这一行命令，就会开始下载外部脚本，但不会等它下载和执行，而是直接执行后面的命令。

defer与async的区别是：前者要等到整个页面正常渲染结束，才会执行；后者一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer是“渲染完再执行”，async是“下载完就执行”。另外，如果有多个defer脚本，会按照它们在页面出现的顺序加载，而多个async脚本是不能保证加载顺序的。