# 整体架构-jQuery3.2.1

# 创建jQuery对象----(L94)

# jQuery原型添加方法与属性

# 核心函数

# 选择器sizzle引擎

# 回调对象

# Deferred对象(异步队列)

# 数据缓存

# 队列操作

# 事件系统

## 概述

* + - 1. 不管是用什么方式绑定,归根到底还是用addEventListener/attachEvent处理的

### jQuery事件系统能给我们解决什么问题

* + - 1. 解决浏览器事件兼容问题
      2. 可以在一个事件类型上添加多个事件处理函数，可以一次添加多个事件类型的事件处理函数
      3. 提供了常用事件的便捷方法
      4. 支持自定义事件
      5. 扩展了组合事件
      6. 提供了统一的事件封装、绑定、执行、销毁机制

### jQuery对事件做了哪些额外工作

* + - 1. 兼容性问题，jQuery.event.fix 对游览器的差异性进行包装处理
         1. 事件对象的获取兼容，IE的event在是在全局的window，标准的是event是事件源参数传入到回调函数中
         2. 目标对象的获取兼容，IE中采用srcElement，标准是target
         3. relatedTarget只是对于mouseout、mouseover有用。在IE中分成了to和from两个Target变量，在mozilla中 没有分开。为了保证兼容，采用relatedTarget统一起来
         4. event的坐标位置兼容
      2. 事件的存储优化
         1. jQuery并没有将事件处理函数直接绑定到DOM元素上，而是通过.data存储在缓存.cahce上

### 自定义事件

* + - 1. 我们在定义组件的时候，浏览器的默认事件往往不能满足我们的要求，比如我们写了一个树形组件，它有一个实例方法init用来完成这个组件的初始化工作，在这个方法调用结束之后，我们通常会自定义一个init事件，以便外部可以在树组件初始化完成之后做一些回调处理：
      2. 自定义事件的使用就跟浏览器默认事件的使用没有任何区别，就连事件冒泡和阻止事件默认行为都完全支持，唯一的区别在于：浏览器自带的事件类型可以通过浏览器的UI线程去触发，而自定义事件必须通过代码来手动触发；即必须调用trigger
      3. 举例

var Tree = function(element, func) {  
 var $tree = this.$tree = $(element);  
 //监听init事件，触发  
 $tree.on('init', func);  
 this.init();  
};  
Tree.prototype.init = function() {  
 console.log('tree init!');  
};  
Tree.prototype.trigger = function () {  
 this.$tree.trigger('init');  
};  
var tree = new Tree('#aa',function () {  
 console.log('aaa完成很多事情');  
 }  
);  
tree.trigger();

### 事件命名空间

* + - 1. jQuery允许在一个元素绑定多个事件监听，如下：但问题是当我们删除doSomething这个事件监听时，应该如何办呢？

$('#element').on('click', doSomething).on('click', doSomethingElse);

* + - 1. 通过命名空间来解决这个问题，因此此时doSomething指定的命名空间为a，doSomethingElse命名空间为b，即使doSomething为匿名函数，也算是函数有了个对应的名字

$('#element').on('click.a', doSomething).on('click.b', doSomethingElse);

* + - 1. 通过带有命名空间的事件，可以解绑对应的事件监听函数

$('#element').off('click.a')

## 事件绑定

### 外部调用函数on与one----(L5569)

* + - 1. 调用方式为$(‘#aaa’).on

on: function( types, selector, data, fn ) {  
 return on( this, types, selector, data, fn );  
},  
 // 为每一个匹配元素的特定事件（像click）绑定一个一次性的事件处理函数。实际就是将fn进行了包装为func，func先进行off事件，然后再调用fn  
one: function( types, selector, data, fn ) {  
 return on( this, types, selector, data, fn, 1 );  
},

### 内部函数on------(L4905)

* + - 1. 此函数先对参数进行调整，然后调用jQuery.event.add添加事件
      2. 多个外部函数调用，如on，one，bind，delegate实际都是调用的此函数

/\*\*  
 \* 事件系统内部函数，on方法实质只完成一些参数调整的工作，然后调用jQuery.event.add添加事件  
 \* @param elem  
 \* @param types 添加到元素的一个或多个事件  
 \* @param selector 一个选择器字符串，用于过滤出被选中的元素中能触发事件的后代元素  
 \* @param data 当一个事件被触发时，要传递给事件处理函数的  
 \* @param fn 事件被触发时，执行的函数  
 \* @param one  
 \* @return {\*}  
 \*/  
function on( elem, types, selector, data, fn, one ) {  
 var origFn, type;  
  
 // types参数可以是一个{types:handlers}对象结构  
 if ( typeof types === "object" ) {  
  
 // 参数形式，types为object对象，( types-Object, selector, data )  
 if ( typeof selector !== "string" ) {  
 // 如果selector不是string，则参数有data，用data，无data，则将selector赋值给data  
 // ( types-Object, data )  
 data = data || selector;  
 selector = undefined;  
 }  
 for ( type in types ) {  
 on( elem, type, selector, data, types[ type ], one );  
 }  
 return elem;  
 }  
 // 调用是 types function，后面无参数  
 if ( data == null && fn == null ) {  
  
 // ( types, fn )  
 fn = selector;  
 data = selector = undefined;  
 } else if ( fn == null ) {  
 if ( typeof selector === "string" ) {  
  
 // ( types, selector, fn )  
 fn = data;  
 data = undefined;  
 } else {  
  
 // ( types, data, fn )  
 fn = data;  
 data = selector;  
 selector = undefined;  
 }  
 }  
 if ( fn === false ) {  
 fn = returnFalse;  
 } else if ( !fn ) {  
 return elem;  
 }  
 // 上面的if主要是对参数进行调整，针对不同调用方式，调整参数  
 // 对于one这种方式调用，将元素fn进行了包装，先off，然后再调用  
 if ( one === 1 ) {  
 origFn = fn;  
 fn = function( event ) {  
  
 // 使用空集合，因为event包含着信息  
 jQuery().off( event );  
 return origFn.apply( this, arguments );  
 };  
  
 // 使用相同的guid，调用者可以使用origFn移除  
 fn.guid = origFn.guid || ( origFn.guid = jQuery.guid++ );  
 }  
 // 当elem未匹配到元素时，each会计算elem长度，故elem.length==0，不会运行添加函数  
 return elem.each( function() {  
 jQuery.event.add( this, types, fn, data, selector );  
 } );  
}

### jQuery.event. add---(L4974 );

* + - 1. html结构

<div class="rex" id="b">  
 <div id="a"></div>  
 <p id="p"></p>  
</div>

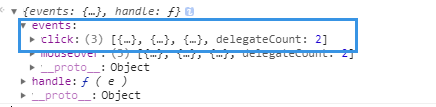
* + - 1. js测试代码

$('#b').on('click mouseover','#a',function () {  
 console.log('a')  
}).on('click mouseover',function () {  
 console.log('b')  
}).on('click mouseover','#p',function () {  
 console.log('p')  
})

* + - 1. 全部运行完elemData的数据格式



* + - * 1. 可以看到存储的数据elemData为event，和一个handle；
        2. Event每一个事件类型对应一个数组



* + - * 1. 每个数组的delegateCount表示，当前元素有几个是事件委托，即实际触发事件在此元素的子元素上
      1. 如何给同一个元素绑定多个事件处理函数而不是覆盖

第一次获取缓存数据elemData，如当前节点elm无缓存数据，则会创建一个；然后分别在elm上创建两个key，handler与event；

handlers = events[ type ]则是在key为event上创建以事件类型type，如click为key的对象，对应的value值则为一个数组，即是事件处理函数；

* + - 1. 源码

/\*\*  
 \* 添加事件,对外调用$().on(types, selector, data, fn );  
 \* add函数参数与外部调用参数顺序有所不同  
 \* 此函数完成的功能：  
 \* 1、添加基本事件，2、添加click mouseover，同时添加多个事件，  
 \* 3、添加自定义事件，  
 \* 4、对有selector添加事件,只是利用jQuery.find.matchesSelector判断传入的selector是否为正确selector表达式，  
 \* 并不保证html中有，之后将selector存储在handleObj  
 \* @param elem 要绑定的jQuery对象  
 \* @param types 事件类型，如click  
 \* @param handler 事件处理函数，如function(){console.log('a');}  
 \* @param data 外部的data数据  
 \* @param selector 选择器  
 \*/  
add: function( elem, types, handler, data, selector ) {  
  
 var handleObjIn, eventHandle, tmp,  
 events, t, handleObj,  
 special, handlers, type, namespaces, origType,  
 // 缓存的数据结构  
 elemData = dataPriv.get( elem );  
  
 // 不要将事件添加到noData或text/comment节点上  
 // ????????????????????????????????不懂何种情况elemData为false  
 if ( !elemData ) {  
 return;  
 }  
  
 // handler可以传递自定义数据对象而不是处理函数，即调用者能通过自定义数据替换handler  
 if ( handler.handler ) {  
 handleObjIn = handler;  
 handler = handleObjIn.handler;  
 selector = handleObjIn.selector;  
 }  
  
 // 确保在添加事件时，无效的选择器就先抛出异常  
 // documentElement 可能就是一个非元素节点  
 if ( selector ) {  
 // 利用Sizzle，看能否再elem上查询到expr这个元素，如不能返回false  
 jQuery.find.matchesSelector( documentElement, selector );  
 }  
  
 // 确保handler有唯一Id，之后用于查询与删除  
 if ( !handler.guid ) {  
 handler.guid = jQuery.guid++;  
 }  
  
 // 如当前元素第一次调用on，则初始化event结构和主处理函数,因此对于同一个elem只会创建一次  
 if ( !( events = elemData.events ) ) {  
 events = elemData.events = {};  
 }  
 // 注意此处的elemData.handle并不是调用时传入的事件处理函数,即，后面addEventListener添加的是这个函数而不是传入的事件处理函数  
 // 因此，jQuery.event.dispatch也是实际触发事件时响应的函数  
 if ( !( eventHandle = elemData.handle ) ) {  
 // 因为eventHandle是实际添加到addEventListener的事件处理函数，故e为原生的事件对象  
 eventHandle = elemData.handle = function( e ) {  
  
 // 当页面卸载后事件被调用，则放弃jQuery.event.trigger()的第二个事  
 return typeof jQuery !== "undefined" && jQuery.event.triggered !== e.type ?  
 // 故arguments为全部的原生事件e  
 jQuery.event.dispatch.apply( elem, arguments ) : undefined;  
 };  
 }  
  
 // 处理由空格分隔的多个事件  
 types = ( types || "" ).match( rnothtmlwhite ) || [ "" ];  
 t = types.length;  
 while ( t-- ) {  
 tmp = rtypenamespace.exec( types[ t ] ) || [];  
 type = origType = tmp[ 1 ];  
 // rtypenamespace = /^([^.]\*)(?:\.(.+)|)/  
 // 如aaa.bb.ccc，则tmp[1]为aaa，temp[2]为bbb.ccc  
 namespaces = ( tmp[ 2 ] || "" ).split( "." ).sort();  
  
 // 对于无命名空间的handlers，必须有一个事件类型type，  
 if ( !type ) {  
 continue;  
 }  
  
 // 如果事件改变其类型，使用special事件处理器来处理更改后的事件类型  
 special = jQuery.event.special[ type ] || {};  
  
 // 如果选择器已定义，确定special事件API类型，否则给他一个类型  
 type = ( selector ? special.delegateType : special.bindType ) || type;  
  
 // 基于新设置的类型更新special  
 special = jQuery.event.special[ type ] || {};  
  
 // handleObj贯穿整个事件处理,绑定事件信息  
 handleObj = jQuery.extend( {  
 type: type,  
 origType: origType,  
 data: data,  
 handler: handler,  
 guid: handler.guid,  
 selector: selector,  
 // Expr = Sizzle.selectors  
 // jQuery.expr = Sizzle.selectors  
 // match: matchExpr  
 needsContext: selector && jQuery.expr.match.needsContext.test( selector ),  
 namespace: namespaces.join( "." )  
 }, handleObjIn );  
  
 // 初次使用，初始化事件handler队列  
 if ( !( handlers = events[ type ] ) ) {  
 handlers = events[ type ] = [];  
 handlers.delegateCount = 0;  
  
 // 非自定义事件，如果special事件处理器返回false，则只能使用addEventListener/attachEvent  
 if ( !special.setup ||  
 special.setup.call( elem, data, namespaces, eventHandle ) === false ) {  
  
 if ( elem.addEventListener ) {  
 elem.addEventListener( type, eventHandle );  
 }  
 }  
 }  
 //自定义事件绑定  
 if ( special.add ) {  
 special.add.call( elem, handleObj );  
  
 if ( !handleObj.handler.guid ) {  
 handleObj.handler.guid = handler.guid;  
 }  
 }  
  
 // 将事件对象handleObj添加到元素的处理列表,代理计数递增  
 // 如有selector,表明当前的事件绑定是委托，这个是将委托事件放在handlers事件列表前面  
 /// 对于如下结构，  
 // <div class="rex" id="b">  
 // <div id="a"></div>  
 // <p id="p"></p>  
 // </div>  
 // 如下代码handlers里面存放的是#a,#p，之后才是console.log(b)这个响应函数  
 // $('#b').on('click mouseover','#a',function () {  
 // console.log('a')  
 // }).on('click mouseover',function () {  
 // console.log('b')  
 // }).on('click mouseover','#p',function () {  
 // console.log('p')  
 // })  
 if ( selector ) {  
 // arrayObject.splice(index,howmany,item1,.....,itemX)  
 // index,整数，规定添加/删除项目的位置，使用负数可从数组结尾处规定位置。  
 // howmany,要删除的项目数量。如果设置为 0，则不会删除项目。  
 // items1...，要添加的内容  
 // handlers后面添加handleObj  
 handlers.splice( handlers.delegateCount++, 0, handleObj );  
 } else {  
 handlers.push( handleObj );// 上面handlers初始化为[]  
 }  
  
 // 追踪使用过的事件，为了事件优化  
 jQuery.event.global[ type ] = true;  
 }  
  
},

## jQuery.Event—对原生event对象进行了封装

### jQuery.Event = function( src, props )---（L5381）

// jQuery.Event构造函数,仅定义一些定义的属性与方法  
jQuery.Event = function( src, props ) {  
  
 // 允许实例化Event，不使用new关键字  
 // 如直接调用var a = jQuery.Event，则this为window对象，或其他，不是jQuery.Event对象  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*提供如何允许实例化Event，不使用new关键字  
 if ( !( this instanceof jQuery.Event ) ) {  
 return new jQuery.Event( src, props );  
 }  
  
 // 构建Event 对象  
 if ( src && src.type ) {  
 this.originalEvent = src;  
 this.type = src.type;  
  
 // 事件冒泡的文档可能被标记为阻止默认事件发生；isDefaultPrevented可以反应是否阻止的标志的正确值  
 this.isDefaultPrevented = src.defaultPrevented ||  
 src.defaultPrevented === undefined &&  
  
 // Support: Android <=2.3 only  
 src.returnValue === false ?  
 returnTrue : // 返回true和false的函数  
 returnFalse;  
  
 // 创建target属性  
 // Support: Safari <=6 - 7 only  
 // Target不应为一个text节点(#504, #13143)  
 this.target = ( src.target && src.target.nodeType === 3 ) ?  
 src.target.parentNode :  
 src.target;  
  
 this.currentTarget = src.currentTarget;  
 this.relatedTarget = src.relatedTarget;  
  
 // src为事件类型  
 } else {  
 this.type = src;  
 }  
  
 // 将明确提供的特征添加到事件对象上  
 if ( props ) {  
 jQuery.extend( this, props );  
 }  
  
 // 创建一个时间戳如果传入的事件不只一个  
 this.timeStamp = src && src.timeStamp || jQuery.now();//jQuery.now = Date.now  
  
 // 标记事件已经修正过  
 this[ jQuery.expando ] = true;  
};

### jQuery.Event.prototype---（L5432）

* + - 1. jQuery.Event原型

// jQuery.Event 基于DOM3事件的  
// 事件对象默认方法的重写，主要是记录是否调用过此方法  
jQuery.Event.prototype = {  
 constructor: jQuery.Event,  
 isDefaultPrevented: returnFalse,  
 isPropagationStopped: returnFalse,  
 isImmediatePropagationStopped: returnFalse,  
 isSimulated: false,  
 // 取消事件的所有默认动作  
 preventDefault: function() {  
 var e = this.originalEvent;  
 // 只是增加了此属性用于记录是否调用过此方法  
 this.isDefaultPrevented = returnTrue;  
  
 if ( e && !this.isSimulated ) {  
 e.preventDefault();  
 }  
 },  
 // 阻止捕获和冒泡阶段中当前事件的进一步传播。  
 stopPropagation: function() {  
 var e = this.originalEvent;  
  
 this.isPropagationStopped = returnTrue;  
  
 if ( e && !this.isSimulated ) {  
 e.stopPropagation();  
 }  
 },  
 // 阻止调用相同事件的其他侦听器。  
 stopImmediatePropagation: function() {  
 var e = this.originalEvent;  
  
 this.isImmediatePropagationStopped = returnTrue;  
  
 if ( e && !this.isSimulated ) {  
 // nativeEvent中，stopImmediatePropagation阻止事件冒泡，该元素绑定的后序相同类型事件的监听函数的执行也将被阻止.  
 e.stopImmediatePropagation();  
 }  
 // 因为stopImmediatePropagation会阻止冒泡，所以需要调用this，记录已经阻止冒泡  
 this.stopPropagation();  
 }  
};

### 为jQuery.Event添加原生事件对象-----(L5470)

* + - 1. 上面jQuery.Event构造函数通过这个jQuery.each将原生对象联系到jQuery.Event上

// 为jQuery.Event添加原生事件对象的属性等  
jQuery.each( {  
 // 这些key是在事件响应时，事件对象所包含的  
 altKey: true,  
 bubbles: true,  
 cancelable: true,  
 changedTouches: true,  
 ctrlKey: true,  
 detail: true,  
 eventPhase: true,  
 metaKey: true,  
 pageX: true,  
 pageY: true,  
 shiftKey: true,  
 view: true,  
 "char": true,  
 charCode: true,  
 key: true,  
 keyCode: true,  
 button: true,  
 buttons: true,  
 clientX: true,  
 clientY: true,  
 offsetX: true,  
 offsetY: true,  
 pointerId: true,  
 pointerType: true,  
 screenX: true,  
 screenY: true,  
 targetTouches: true,  
 toElement: true,  
 touches: true,  
 // 利用which判断点下的是哪个键  
 // 原生事件中， MouseEvent.which 表示鼠标按下的是哪个键，（非标准）  
 which: function( event ) {  
 var button = event.button;  
  
 // 为which增加key的事件  
 if ( event.which == null && rkeyEvent.test( event.type ) ) {  
 // charCode : 字符键的字符Unicode值(非标准值);MDN上标注为废弃属性，用event.key代替  
 // keyCode ：获取按下的键盘按键Unicode值，MDN上标注为废弃属性  
 return event.charCode != null ? event.charCode : event.keyCode;  
 }  
  
 // Add which for click: 1 === left; 2 === middle; 3 === right  
 if ( !event.which && button !== undefined && rmouseEvent.test( event.type ) ) {  
 if ( button & 1 ) {  
 return 1;  
 }  
  
 if ( button & 2 ) {  
 return 3;  
 }  
  
 if ( button & 4 ) {  
 return 2;  
 }  
  
 return 0;  
 }  
  
 return event.which;  
 }  
}, jQuery.event.addProp );

### jQuery.event. addProp----(L5284)

* + - 1. 为jQuery.Event.prototype添加原生事件的属性具体实现函数

addProp: function( name, hook ) {  
 Object.defineProperty( jQuery.Event.prototype, name, {  
 enumerable: true,  
 configurable: true,  
  
 get: jQuery.isFunction( hook ) ?  
 function() {  
 if ( this.originalEvent ) {  
 return hook( this.originalEvent );  
 }  
 } :  
 function() {  
 if ( this.originalEvent ) {  
 return this.originalEvent[ name ];  
 }  
 },  
  
 set: function( value ) {  
 Object.defineProperty( this, name, {  
 enumerable: true,  
 configurable: true,  
 writable: true,  
 value: value  
 } );  
 }  
 } );  
},

## 事件处理流程

### jQuery.event.handlers: function( event, handlers )----(L5226)

/\*\*  
 \* 将有序地返回当前事件所需执行的所有事件处理程序。  
 \* 主要是将委托事件与自身事件进行顺序调整，形成队列，  
 \* 因为冒泡是有顺序的，故父节点的事件应该在子节点事件之后响应  
 \* @param event jquery 包装后的event对象  
 \* @param handlers  
 \* @return {Array} 委托的事件处理程序相对于直接绑定的事件处理程序在队列的更前面，委托层次越深，该事件处理程序则越靠前  
 \*/  
handlers: function( event, handlers ) {  
 var i, handleObj, sel, matchedHandlers, matchedSelectors,  
 handlerQueue = [],  
 delegateCount = handlers.delegateCount,  
 cur = event.target; // 当前触发事件的对象  
  
 // 寻找委托事件，delegateCount!=0 才表示有委托事件  
 if ( delegateCount &&  
 // 解决某些浏览器的bug问题  
 // Support: IE <=9  
 // 目标对象是元素节点(trac-13180)  
 cur.nodeType &&  
  
 // Support: Firefox <=42  
 // 点击的不是鼠标主按键(trac-3861)  
 // https://www.w3.org/TR/DOM-Level-3-Events/#event-type-click  
 // Support: IE 11 only  
 // 对于非鼠标输入工具，button值可能为-1(gh-2343)  
 !( event.type === "click" && event.button >= 1 ) ) {  
 // 主要使用event.target事件源节点不断循环往上查找父节点，  
 // 看些节点和是否在handlers中的选择器对应的节点中  
 // 如子父级关系是，#a<#b<#c,其中#a为事件触发target，#c为事件绑定对象  
 // cur每次循环分别为#a<#b<#c，但handlers有3个事件函数，每次循环，都需要循环handlers（长度由delegateCount决定），  
 // 判断是否为当前节点的事件处理函数,因为外部并没有判断selector是否能查询到节点  
 //  
 for ( ; cur !== this; cur = cur.parentNode || this ) {  
  
 // 不检测非元素节点 (#13208)  
 // 不要对禁用元素进行单击处理(#6911, #8165, #11382, #11764)  
 if ( cur.nodeType === 1 && !( event.type === "click" && cur.disabled === true ) ) {  
 matchedHandlers = [];  
 matchedSelectors = {};  
 for ( i = 0; i < delegateCount; i++ ) {// 遍历委托元素  
 handleObj = handlers[ i ];  
  
 // 避免与Object.prototype属性发生冲突 (#13203)  
 sel = handleObj.selector + " ";  
  
 if ( matchedSelectors[ sel ] === undefined ) {// 如能查到push到matchedHandlers中  
 matchedSelectors[ sel ] = handleObj.needsContext ?  
 jQuery( sel, this ).index( cur ) > -1 : // 如是同代元素  
 jQuery.find( sel, this, null, [ cur ] ).length;//jquery.find 这个函数是找出正在处理的元素的后代元素的好方法  
 }  
 if ( matchedSelectors[ sel ] ) {  
 matchedHandlers.push( handleObj );  
 }  
 }  
 // 如委托事件都能找到target，则matchedHandlers.length>0，则push到handlerQueue中  
 if ( matchedHandlers.length ) {  
 handlerQueue.push( { elem: cur, handlers: matchedHandlers } );  
 }  
 }  
 }  
 }  
  
 // 添加非委托的事件，即this本身的事件处理函数  
 cur = this;  
 if ( delegateCount < handlers.length ) { // 如为false，则表明都是委托事件，从delegateCount后面的handlers都是本身事件处理函数  
 handlerQueue.push( { elem: cur, handlers: handlers.slice( delegateCount ) } );  
 }  
  
 return handlerQueue;  
},

### jQuery.event.dispatch: function( nativeEvent )----(L5162)

/\*\*  
 \* 当事件响应时，会触发此函数  
 \* @param nativeEvent 原生的事件对象  
 \* @return {undefined|\*}  
 \*/  
dispatch: function( nativeEvent ) {  
  
 // 由原生对象构建一个可写的jQuery.Event  
 var event = jQuery.event.fix( nativeEvent );  
  
 var i, j, ret, matched, handleObj, handlerQueue,  
 args = new Array( arguments.length ),  
 // 当前对象在dataPriv中存储的两个key，一个是events，另一个是handler  
 // 获取当前节点缓存中对应事件类型的事件处理列表  
 handlers = ( dataPriv.get( this, "events" ) || {} )[ event.type ] || [],  
 special = jQuery.event.special[ event.type ] || {};  
  
 // 使用jQuery.Event而不是native事件  
 args[ 0 ] = event;  
 // 将原生事件的其他参数全部复制到args上  
 for ( i = 1; i < arguments.length; i++ ) {  
 args[ i ] = arguments[ i ];  
 }  
 // delegateTarget为jQuery.event.special里面具体事件的一个属性  
 event.delegateTarget = this;  
  
 // 调用preDispatch钩子方法，如为true，则直接返回  
 if ( special.preDispatch && special.preDispatch.call( this, event ) === false ) {  
 return;  
 }  
  
 // 确定handler队列,获取到符合要求的委托处理函数队列  
 handlerQueue = jQuery.event.handlers.call( this, event, handlers );  
  
 // 针对handlerQueue的筛选  
 i = 0;  
 // 事件未PropagationStop,则继续  
 while ( ( matched = handlerQueue[ i++ ] ) && !event.isPropagationStopped() ) {  
 // ???????????????????????????????????为何要交换currentTarget，不影响propagation  
 event.currentTarget = matched.elem;  
  
 j = 0;  
 // 循环每个事件类型的多个事件处理函数  
 while ( ( handleObj = matched.handlers[ j++ ] ) &&  
 !event.isImmediatePropagationStopped() ) {  
  
 // 触发事件的条件：  
 // 1)没有命名空间，或  
 // 2)有命名空间的子集或等于那些边界事件（他们两者都可以没有命名空间）  
 // 设置一些触发条件，并不是什么时候都能触发事件  
 if ( !event.rnamespace || event.rnamespace.test( handleObj.namespace ) ) {  
  
 event.handleObj = handleObj;  
 event.data = handleObj.data;  
  
 ret = ( ( jQuery.event.special[ handleObj.origType ] || {} ).handle ||  
 handleObj.handler ).apply( matched.elem, args );  
  
 if ( ret !== undefined ) {  
 if ( ( event.result = ret ) === false ) {  
 event.preventDefault();  
 event.stopPropagation();  
 }  
 }  
 }  
 }  
 }  
  
 // 针对mapped类型，调用postDispatch钩子方法  
 if ( special.postDispatch ) {  
 special.postDispatch.call( this, event );  
 }  
  
 return event.result;  
},

## 触发指定事件(trigger)

### 对外接口----(L8112)

// 触发事件的对外接口$().trigger();  
jQuery.fn.extend( {  
 // 在每一个匹配的元素上触发某类事件。  
 trigger: function( type, data ) {  
 return this.each( function() {  
 jQuery.event.trigger( type, data, this );  
 } );  
 },  
 // 与trigger区别  
 // 第一，他不会触发浏览器默认事件。如$().triggerHandler('focus');只会触发绑定的函数，浏览器默认事件不触发  
 // 第二，只触发jQuery对象集合中第一个元素的事件处理函数。  
 // 第三，这个方法的返回的是事件处理函数的返回值，而不是据有可链性的jQuery对象。此外，如果最开始的jQuery对象集合为空，则这个方法返回 undefined  
 triggerHandler: function( type, data ) {  
 var elem = this[ 0 ];  
 if ( elem ) { // 只处理jQuery对象集合的第一个元素  
 return jQuery.event.trigger( type, data, elem, true );  
 }  
 }  
} );

### jQuery.event.trigger----(L8114)

/\*\*  
 \* 触发事件内部API  
 \* 模拟事件触发,为了让事件模型在各浏览器上表现一致 (并不推荐使用)  
 \* @param {Object} event 事件对象 (原生Event事件对象将被转化为jQuery.Event对象)  
 \* @param {Object} data 自定义传入到事件处理函数的数据  
 \* @param {Object} elem HTML Element元素  
 \* @param {Boolen} onlyHandlers 是否不冒泡 true 表示不冒泡 false表示冒泡；用于判断是否阻止浏览器默认行为  
 \*/  
trigger: function( event, data, elem, onlyHandlers ) {  
  
 var i, cur, tmp, bubbleType, ontype, handle, special,  
 eventPath = [ elem || document ],  
 // 判断某个对象是否含有指定的属性，不会在原型链上查找  
 type = hasOwn.call( event, "type" ) ? event.type : event,  
 namespaces = hasOwn.call( event, "namespace" ) ? event.namespace.split( "." ) : [];// 事件是否有命名空间，有则分割成数组  
  
 cur = tmp = elem = elem || document;  
  
 // 对于text和comment节点不进行事件处理  
 if ( elem.nodeType === 3 || elem.nodeType === 8 ) {  
 return;  
 }  
  
 // 仅对focus/blur事件变种成focusin/out进行处理  
 // 如果浏览器原生支持focusin/out，则确保当前不触发他们  
 // ????????????????????????????不知如何判断，jQuery.event.triggered开始时为undefined  
 if ( rfocusMorph.test( type + jQuery.event.triggered ) ) {  
 return;  
 }  
 // 如果type有命名空间，命名空间的过滤  
 // 只要有.则表示有命名空间，.第一个为事件类型，后面的都是命名空间  
 if ( type.indexOf( "." ) > -1 ) {  
  
 // 有命名空间的trigger调用，在handle()函数中会构建一个正则去匹配  
 namespaces = type.split( "." );  
 type = namespaces.shift();  
 namespaces.sort(); // 因为add时，namespaces调用了sort，故此处调用sort保证顺序一致  
 }  
 // 检测是否需要改成ontype形式 即"onclick"  
 ontype = type.indexOf( ":" ) < 0 && "on" + type;  
  
 // jQuery.expando:检测事件对象是否由jQuery.Event生成的实例，否则用jQuery.Event改造  
 // 因为调用trigger时的事件类型可能并未绑定过  
 event = event[ jQuery.expando ] ?  
 event :  
 new jQuery.Event( type, typeof event === "object" && event );  
  
 // 对event预处理  
 // Trigger bitmask: & 1 for native handlers; & 2 for jQuery (always true)  
 event.isTrigger = onlyHandlers ? 2 : 3;  
 event.namespace = namespaces.join( "." );// sort后再拼接  
 // 如namespaces=['a','b'];正则：/(^|\.)a\.(?:.\*\.|)b(\.|$)/  
 event.rnamespace = event.namespace ?  
 new RegExp( "(^|\\.)" + namespaces.join( "\\.(?:.\*\\.|)" ) + "(\\.|$)" ) :  
 null;  
  
 // 清除事件返回数据，以重新使用  
 event.result = undefined;  
 // 如果事件没有触发元素，则用elem代替  
 if ( !event.target ) {  
 event.target = elem;  
 }  
  
 // 克隆任何输入数据并预设event，构建handler 参数列表  
 // 如果data为空，则传入处理函数的是event，否则由data和event组成  
 data = data == null ?  
 [ event ] :  
 jQuery.makeArray( data, [ event ] );  
  
 // 尝试通过特殊事件进行处理，必要时候退出函数  
 special = jQuery.event.special[ type ] || {};  
 // 对于blur，focus，jQuery.event.special存储了事件原生的事件处理函数  
 if ( !onlyHandlers && special.trigger && special.trigger.apply( elem, data ) === false ) {  
 return;  
 }  
 // 模拟事件冒泡  
 // 根据W3C规范，需要预设事件传播路径 (#9951)  
 // 事件冒泡到document，然后是window，之后寻找全局var (#9724)  
 // trigger与triggerHandler的本质区别实现在这里了,triggerHandler不会进入这里面  
 // 故eventPath只是当前元素的列表  
 if ( !onlyHandlers && !special.noBubble && !jQuery.isWindow( elem ) ) {  
 // 冒泡时是否需要转成别的事件(用于事件模拟)  
 bubbleType = special.delegateType || type;  
 // 如果不是变形来的foucusin/out事件  
 if ( !rfocusMorph.test( bubbleType + type ) ) {  
 cur = cur.parentNode;  
 }  
 // 遍历自身及所有父节点放在eventPath里面  
 for ( ; cur; cur = cur.parentNode ) {  
 eventPath.push( cur );  
 tmp = cur;  
 }  
  
 // 当到达document时，将window添加到eventPath中  
 if ( tmp === ( elem.ownerDocument || document ) ) {  
 // defaultView :在浏览器中，该属性返回当前 document 对象所关联的 window 对象，如果没有，会返回 null。  
 eventPath.push( tmp.defaultView || tmp.parentWindow || window );  
 }  
 }  
  
 // 触发所有事件监听函数  
 i = 0;  
 while ( ( cur = eventPath[ i++ ] ) && !event.isPropagationStopped() ) {  
  
 event.type = i > 1 ?  
 bubbleType :  
 special.bindType || type;  
  
 // 检测数据缓存中是否有此事件类型，如有取出handle  
 handle = ( dataPriv.get( cur, "events" ) || {} )[ event.type ] &&  
 dataPriv.get( cur, "handle" );  
 if ( handle ) {  
 // 触发handle  
 handle.apply( cur, data );  
 }  
  
 // 原生handler  
 // 取出原生事件处理器elem.ontype (比如click事件就是elem.onclick)  
 handle = ontype && cur[ ontype ];  
 // acceptData:判断绑定数据的目标owner类型是否符合  
 if ( handle && handle.apply && acceptData( cur ) ) {  
 event.result = handle.apply( cur, data );  
 if ( event.result === false ) {  
 event.preventDefault();  
 }  
 }  
 }  
 // 保存事件类型，因为这时候事件可能变了  
 event.type = type;  
  
 // 如果不需要阻止默认动作，立即执行  
 if ( !onlyHandlers && !event.isDefaultPrevented() ) {  
  
 if ( ( !special.\_default ||  
 special.\_default.apply( eventPath.pop(), data ) === false ) &&  
 acceptData( elem ) ) {  
  
 // 在目标上调用与事件同名的原生DOM方法。  
 // 不能仅用jQuery.isFunction检测，因为ie6,7可能失败 (#6170)  
 // 确保不对window对象阻止默认事件  
 if ( ontype && jQuery.isFunction( elem[ type ] ) && !jQuery.isWindow( elem ) ) {  
  
 // 调用FOO()时，避免重复触发onFOO事件  
 tmp = elem[ ontype ];  
 // 清除掉该事件监听  
 if ( tmp ) {  
 elem[ ontype ] = null;  
 }  
  
 // 当我们已经将事件向上起泡时，防止相同事件再次触发  
 jQuery.event.triggered = type;  
 // HTMLElement原生事件处理函数触发  
 elem[ type ]();  
 // 完成清除标记  
 jQuery.event.triggered = undefined;  
 // 事件触发完了，可以把监听重新绑定回去  
 if ( tmp ) {  
 elem[ ontype ] = tmp;  
 }  
 }  
 }  
 }  
  
 return event.result;  
},

## 特殊事件处理

### 概述

* + - 1. 委托设计是基于事件可冒泡的。但是有些事件是不可冒泡的，有的事件在不同的浏览器上支持的冒泡情况不同。还有不同的浏览器支持的事件类型也不尽相同。这些处理主要都被放在jQuery.event.special中。
      2. jQuery.event.special对象中保存着为适配特定事件所需的变量和方法。
      3. 主要是将没有事件冒泡机制，使用某种方式也能进行委托设计
      4. jQuery3.2，支持是ie9+，故submit，change已经是冒泡的了，不需要做兼容

### jQuery .event.special----(L5318)

* + - 1. 对于不能冒泡的事件，在此处进行转换，如绑定了focus，这里将focus事件改为绑定为focusin，即绑定的实际是delegateType

special: {  
 load: {  
  
 // 阻止触发image.load事件冒泡到window.load  
 noBubble: true  
 },  
 focus: {  
  
 // 触发本当前节点blur/focus事件 确保队列正确  
 trigger: function() {  
 // safeActiveElement,ie<9,document.activeElement会出问题  
 // document.activeElement属性返回文档中当前获得焦点的元素  
 if ( this !== safeActiveElement() && this.focus ) {  
 this.focus();  
 return false;  
 }  
 },  
 delegateType: "focusin"  
 },  
 blur: {  
 trigger: function() {  
 if ( this === safeActiveElement() && this.blur ) {  
 this.blur();  
 return false;  
 }  
 },  
 // 主要标识，是否需要转换为其他事件，即将blur转换为focusout  
 delegateType: "focusout"  
 },  
 click: {  
  
 // 对于chekbox，如选中状态是正确的则触发原生事件  
 trigger: function() {  
 if ( this.type === "checkbox" && this.click && nodeName( this, "input" ) ) {  
 this.click();  
 return false;  
 }  
 },  
  
 // 为了保证跨浏览器兼容性，不在links上触发元素click事件处理函数  
 \_default: function( event ) {  
 return nodeName( event.target, "a" );  
 }  
 },  
  
 beforeunload: {  
 postDispatch: function( event ) {  
  
 // Support: Firefox 20+  
 // 如returnValue未设置，则firefox不会alert  
 if ( event.result !== undefined && event.originalEvent ) {  
 event.originalEvent.returnValue = event.result;  
 }  
 }  
 }  
}

### focus与blur处理

* + - 1. jQuery.event.special.focusin,jQuery.event.special.focuout------(L8304)
         1. 此处主要是解决浏览器无focusin与focusout事件
         2. 为jQuery.event.special扩展focuin与focusout事件
         3. focusin 与focusout 在元素获得或事件焦点时触发，相比之下，focus与blur事件在焦点改变时触发，但不冒泡

support.focusin = "onfocusin" in window; // onfocusin 事件在一个元素即将获得焦点时触发。与focus区别是支持冒泡

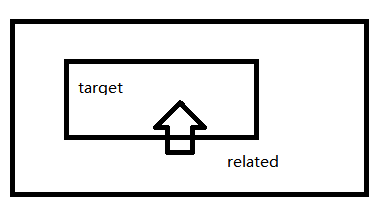
// Support: Firefox <=44  
// 将focus绑定的事件转化为focusin来绑定，focusin在W3C的标准中是冒泡的，除开火狐之外的浏览器也确实支持冒泡  
// firefox不支持focus(in | out)事件- https://bugzilla.mozilla.org/show\_bug.cgi?id=687787  
//  
// Support: Chrome <=48 - 49, Safari <=9.0 - 9.1  
// focus(in | out) 事件在focus & blur events之后触发,  
// 这违背了规范 - http://www.w3.org/TR/DOM-Level-3-Events/#events-focusevent-event-order  
// 这个bug详细在 - https://bugs.chromium.org/p/chromium/issues/detail?id=449857  
// 构建focusin与focuout事件  
if ( !support.focusin ) {  
 jQuery.each( { focus: "focusin", blur: "focusout" }, function( orig, fix ) {  
  
 // 为focusin/focuout附加一个handler  
 var handler = function( event ) {  
 jQuery.event.simulate( fix, event.target, jQuery.event.fix( event ) );  
 };  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // 事件不存在冒泡，但是捕获阶段是从document到指定元素的，故可以在捕获阶段进行绑定  
 jQuery.event.special[ fix ] = {  
 setup: function() {  
 var doc = this.ownerDocument || this,  
 attaches = dataPriv.access( doc, fix );  
 // 因为第一个参数是是orig,因为火狐不支持focusin/focusout所以jQuery使用focus/blur替代来监听事件；  
 // 第三个参数为true，表示在事件捕获阶段触发事件  
 if ( !attaches ) {  
 doc.addEventListener( orig, handler, true );  
 }  
 dataPriv.access( doc, fix, ( attaches || 0 ) + 1 );  
 },  
 // 删除事件绑定  
 teardown: function() {  
 var doc = this.ownerDocument || this,  
 attaches = dataPriv.access( doc, fix ) - 1;  
  
 if ( !attaches ) {  
 doc.removeEventListener( orig, handler, true );  
 dataPriv.remove( doc, fix );  
  
 } else {  
 dataPriv.access( doc, fix, attaches );  
 }  
 }  
 };  
 } );  
}

* + - 1. jQuery.event.simulate---------(L8251)

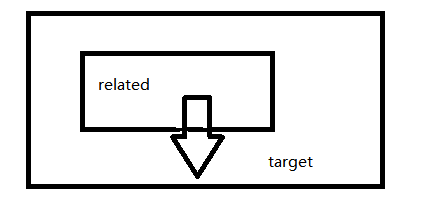
// 用于模拟focus事件  
// 仅仅在`focus(in | out)` 事件上使用，就是对事件e进行改变  
simulate: function( type, elem, event ) {  
 var e = jQuery.extend(  
 new jQuery.Event(),  
 event,  
 {  
 type: type,  
 isSimulated: true  
 }  
 );  
  
 jQuery.event.trigger( e, null, elem );  
}

### mouseenter/mouseleave处理----(L5531)

* + - 1. ( related !== target && !jQuery.contains( target, related ) )为关键判断，假设处理的是mouseenter
      2. 鼠标从related到target，很明显related在target外面，所以当鼠标移动到target的时候满足条件，调用处理



* + - 1. 但如related在里面，target在外面，则对于target来说，一直处于mouseenter状态，故不应该触发mouseenter处理函数



// 使用mouseover/out和事件时机检测创建mouseenter/leave事件  
// 故此种事件可以变为事件委托  
// 用相同方式处理 pointerenter/pointerleave and pointerover/pointerout  
//  
// Support: Safari 7 only  
// Safari 触发mouseenter过于频繁:  
// https://bugs.chromium.org/p/chromium/issues/detail?id=470258  
// 老的chrome也存在这个bug  
jQuery.each( {  
 mouseenter: "mouseover",  
 mouseleave: "mouseout",  
 pointerenter: "pointerover",  
 pointerleave: "pointerout"  
}, function( orig, fix ) {  
 jQuery.event.special[ orig ] = {  
 delegateType: fix,  
 bindType: fix,  
  
 handle: function( event ) {  
 var ret,  
 target = this,  
 related = event.relatedTarget,  
 handleObj = event.handleObj;  
  
 // 对于 mouseenter/leave 事件调用处理函数handler，只有在related元素在target外面  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // 参考: 当鼠标离开/进入浏览器窗口的时候是没有relatedTarget的  
 if ( !related || ( related !== target && !jQuery.contains( target, related ) ) ) {  
 event.type = handleObj.origType;  
 ret = handleObj.handler.apply( this, arguments );  
 event.type = fix;  
 }  
 return ret;  
 }  
 };  
} );

## hover事件-----(L8295)

* + - 1. 提供了一个table每一列悬浮时的一个特效，比如

<p>asdasdasdasdasdasd</p>  
<p>asdasdasdasdasdasd</p>  
<p>asdasdasdasdasdasd</p>  
<p>asdasdasdasdasdasd</p>  
<p>asdasdasdasdasdasd</p>

<script>  
 $('p').hover(  
 function () {  
 $(this).addClass("cur");  
 },  
 function () {  
 $(this).removeClass("cur");  
 }  
 )  
</script>

// 当鼠标移动到一个匹配的元素上面时，会触发指定的第一个函数。当鼠标移出这个元素时，会触发指定的第二个函数。  
jQuery.fn.extend( {  
 hover: function( fnOver, fnOut ) {  
 return this.mouseenter( fnOver ).mouseleave( fnOut || fnOver );  
 }  
} );

## 事件的快捷绑定/调用方式-----(L8282)

* + - 1. 提供类似$(‘#a’).click()的事件触发方式，与$(‘#a’).click(function (){})的函数绑定方式

jQuery.each( ( "blur focus focusin focusout resize scroll click dblclick " +  
 "mousedown mouseup mousemove mouseover mouseout mouseenter mouseleave " +  
 "change select submit keydown keypress keyup contextmenu" ).split( " " ),  
 function( i, name ) {  
  
 // handle事件处理函数绑定，有参数则绑定事件，无参数则直接trigger  
 jQuery.fn[ name ] = function( data, fn ) {  
 return arguments.length > 0 ?  
 this.on( name, null, data, fn ) :  
 this.trigger( name );  
 };  
} );

## 事件解绑

### 对外接口-----(L5575)

// 在选择元素上移除一个或多个事件的事件处理函数。  
off: function( types, selector, fn ) {  
 var handleObj, type;  
 // 前面if所做的工作也是参数调整，与on类似，只是没有on函数兼容的多，off函数不能传入data数据  
 if ( types && types.preventDefault && types.handleObj ) {  
  
 // ( event ) dispatched jQuery.Event  
 handleObj = types.handleObj;  
 jQuery( types.delegateTarget ).off(  
 handleObj.namespace ?  
 handleObj.origType + "." + handleObj.namespace :  
 handleObj.origType,  
 handleObj.selector,  
 handleObj.handler  
 );  
 return this;  
 }  
 // 如果传入tpye为对象，即{click：function,hover:function}  
 // fucntion为之前on绑定的函数名  
 if ( typeof types === "object" ) {  
  
 // ( types-object [, selector] )  
 for ( type in types ) {  
 this.off( type, selector, types[ type ] );  
 }  
 return this;  
 }  
 // 对于$().off(click,function)参数进行调整，  
 if ( selector === false || typeof selector === "function" ) {  
  
 // ( types [, fn] )  
 fn = selector;  
 selector = undefined;  
 }  
 // 如fn === false，则返回一个函数returnFalse  
 if ( fn === false ) {  
 fn = returnFalse;  
 }  
 return this.each( function() {  
 jQuery.event.remove( this, types, fn, selector );  
 } );  
}

### jQuery.event.remove---------(L5088)

// 从element上删除一个或一组事件  
remove: function( elem, types, handler, selector, mappedTypes ) {  
  
 var j, origCount, tmp,  
 events, t, handleObj,  
 special, handlers, type, namespaces, origType,  
 elemData = dataPriv.hasData( elem ) && dataPriv.get( elem );  
  
 // 如当前元素elem无缓存数据，或者无events这个key直接返回  
 if ( !elemData || !( events = elemData.events ) ) {  
 return;  
 }  
  
 // 分解types为type.namespace为单位元素的数组  
 // 即，type可以传入 clikc.name1 hover.name2  
 types = ( types || "" ).match( rnothtmlwhite ) || [ "" ];  
 t = types.length;  
 // 可能同时解绑多个事件类型  
 while ( t-- ) {  
 tmp = rtypenamespace.exec( types[ t ] ) || [];  
 type = origType = tmp[ 1 ];  
 namespaces = ( tmp[ 2 ] || "" ).split( "." ).sort();  
  
 // 解绑当前元素的全部事件，如提供命名空间，则删除当前命名空间下的全部事件  
 if ( !type ) {  
 for ( type in events ) {  
 jQuery.event.remove( elem, type + types[ t ], handler, selector, true );  
 }  
 continue;  
 }  
 // 如果事件改变其类型，使用special事件处理器来处理更改后的事件类型  
 special = jQuery.event.special[ type ] || {};  
 // 如果选择器已定义，确定special事件API类型，否则给他一个类型  
 type = ( selector ? special.delegateType : special.bindType ) || type;  
 handlers = events[ type ] || [];  
 //如namespaces=['a','b'];正则：/(^|\.)a\.(?:.\*\.|)b(\.|$)/  
 tmp = tmp[ 2 ] &&  
 new RegExp( "(^|\\.)" + namespaces.join( "\\.(?:.\*\\.|)" ) + "(\\.|$)" );  
  
 // 删除匹配事件  
 // 一个事件类型可能绑定了多个事件处理函数  
 origCount = j = handlers.length;  
 while ( j-- ) {  
 handleObj = handlers[ j ];  
 //各种满足移除事件的条件才能移除  
 if ( ( mappedTypes || origType === handleObj.origType ) &&  
 ( !handler || handler.guid === handleObj.guid ) &&  
 ( !tmp || tmp.test( handleObj.namespace ) ) &&  
 ( !selector || selector === handleObj.selector ||  
 selector === "\*\*" && handleObj.selector ) ) {  
 // 删除这个handlers  
 handlers.splice( j, 1 );  
 // 如果handler是委托，将委托计数减1  
 if ( handleObj.selector ) {  
 handlers.delegateCount--;  
 }  
 if ( special.remove ) {  
 special.remove.call( elem, handleObj );  
 }  
 }  
 }  
  
 // 如我我们删除东西，并无更多的handlers存在，则删除一般事件处理函数  
 // (避免在删除特殊事件处理程序时进行无限递归。)  
 if ( origCount && !handlers.length ) {  
 if ( !special.teardown ||  
 special.teardown.call( elem, namespaces, elemData.handle ) === false ) {  
  
 jQuery.removeEvent( elem, type, elemData.handle );  
 }  
  
 delete events[ type ];  
 }  
 }  
  
 // 如events为空，则删除缓存的数据  
 if ( jQuery.isEmptyObject( events ) ) {  
 dataPriv.remove( elem, "handle events" );  
 }  
},