# 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

# DOM操作

## 概述

* + - 1. DOM操作包括append、prepend、before、after、replaceWith、appendTo、prependTo、insertBefore、insertAfter、replaceAll。其核心处理函数是domManip。
      2. 浏览器原生的插入节点的方法有两个：appendChild和inserBefore，jQuery利用这两个方法拓展

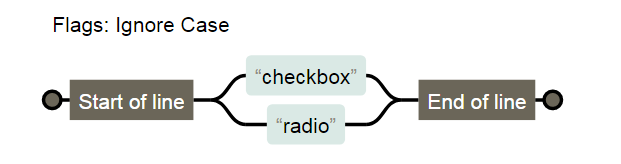
## 克隆节点

### 什么时候需要克隆节点

* + - 1. 在对DOM操作过程中如果直接使用节点会出现节点随操作而变动的情况。
      2. 比如对节点使用.after/.before/.append等方法后，节点被添加到新的地方，原来的位置上的节点被移除了。有的时候需要保留原来位置上的节点，仅仅是需要一个副本添加到对应位置，这个时候克隆就有了使用场景。

### rcheckableType-----(L4693);

* + - 1. var rcheckableType = ( /^(?:checkbox|radio)$/i );
      2. 判断是checkbox或radio



### function fixInput( src, dest ) -----(L5965)

* + - 1. 修复办法就是将克隆有问题的，手动从srcElements复制到destElements

// 修复IE bugs，可以看support测试  
function fixInput( src, dest ) {  
 var nodeName = dest.nodeName.toLowerCase();  
  
 // 克隆的checkbox或radio不能持久化checked状态  
 if ( nodeName === "input" && rcheckableType.test( src.type ) ) {  
 dest.checked = src.checked;  
  
 // 克隆options，不能将所选选项返回到默认选定状态  
 } else if ( nodeName === "input" || nodeName === "textarea" ) {  
 dest.defaultValue = src.defaultValue;  
 }  
}

### function cloneCopyEvent( src, dest )------------(L5661)

/\*\*  
 \* cloneCopyEvent函数中会将原节点的数据保存到克隆节点中，然后将原节点的事件绑定到新的克隆节点上  
 \* 主要是利用数据缓存系统data，分别将dataPriv和dataUser克隆到clone对象  
 \* 注意点是：在克隆事件时，为了保证克隆的事件没有重复，清空了events和handler  
 \* @param src 被克隆的对象  
 \* @param dest clone对象  
 \*/  
function cloneCopyEvent( src, dest ) {  
 var i, l, type, pdataOld, pdataCur, udataOld, udataCur, events;  
 // clone对象非元素节点，return  
 if ( dest.nodeType !== 1 ) {  
 return;  
 }  
  
 // 1. 复制事件的私有数据：events，handlers，等  
 if ( dataPriv.hasData( src ) ) {  
 // 绑定的数据并不一定只有events，handle，还可能有其他私有数据，所以需要先从dataPriv取出src数据  
 pdataOld = dataPriv.access( src );  
 pdataCur = dataPriv.set( dest, pdataOld );  
 events = pdataOld.events;  
 // 如被克隆元素src有events  
 if ( events ) {  
 //保证被克隆的节点的事件对象干净，确保没有后面添加的事件没有重复  
 delete pdataCur.handle;  
 pdataCur.events = {};  
  
 for ( type in events ) {  
 for ( i = 0, l = events[ type ].length; i < l; i++ ) {  
 jQuery.event.add( dest, type, events[ type ][ i ] );  
 }  
 }  
 }  
 }  
  
 // 2. 复制user数据  
 if ( dataUser.hasData( src ) ) {  
 udataOld = dataUser.access( src );  
 udataCur = jQuery.extend( {}, udataOld );  
  
 dataUser.set( dest, udataCur );  
 }  
}

### function setGlobalEval( elems, refElements )------(L4749)

// 全局性地标记scripts代码段已经被执行过了  
// 为每个元素绑定私有数据，key：globalEval，value  
function setGlobalEval( elems, refElements ) {  
 var i = 0,  
 l = elems.length;  
  
 for ( ; i < l; i++ ) {  
 dataPriv.set(  
 elems[ i ],  
 "globalEval",  
 !refElements || dataPriv.get( refElements[ i ], "globalEval" )  
 );  
 }  
}

### $.clone()-------------(L5826)

/\*\*  
 \* 克隆匹配的DOM元素并且选中这些克隆的副本。  
 \* @param elem  
 \* @param dataAndEvents 是否同时复制元素的附加数据和绑定事件  
 \* @param deepDataAndEvents 是否同时复制元素的所有子元素的附加数据和绑定事件  
 \* @return {Node}  
 \*/  
clone: function( elem, dataAndEvents, deepDataAndEvents ) {  
 var i, l, srcElements, destElements,  
 clone = elem.cloneNode( true ),  
 inPage = jQuery.contains( elem.ownerDocument, elem );  
  
 // 修复IE克隆问题  
 if ( !support.noCloneChecked && ( elem.nodeType === 1 || elem.nodeType === 11 ) &&  
 !jQuery.isXMLDoc( elem ) ) {  
  
 // 此处由于性能问题避开使用Sizzle: https://jsperf.com/getall-vs-sizzle/2  
 destElements = getAll( clone ); //根据某个tag名获取context全部tag元素  
 srcElements = getAll( elem );  
 // 针对IE bugs，修复，即将有问题的地方，再从srcElements复制到destElements  
 for ( i = 0, l = srcElements.length; i < l; i++ ) {  
 fixInput( srcElements[ i ], destElements[ i ] );  
 }  
 }  
  
 // 如果要克隆缓存数据（包括普通数据和绑定事件），克隆之  
 if ( dataAndEvents ) {  
 // 如需要克隆子元素的数据和事件，克隆之  
 if ( deepDataAndEvents ) {  
 srcElements = srcElements || getAll( elem );  
 destElements = destElements || getAll( clone );  
  
 for ( i = 0, l = srcElements.length; i < l; i++ ) {  
 cloneCopyEvent( srcElements[ i ], destElements[ i ] );  
 }  
 } else {  
 cloneCopyEvent( elem, clone );  
 }  
 }  
  
 // 保护script计算历史（全局性地标记scripts代码段已经被执行过了），并回收内存，返回克隆节点。  
 // 如果该 script 已经执行过，则不会再次执行。或者说执行后会设置一个 globalEval: true 的标示。

destElements = getAll( clone, "script" );  
 if ( destElements.length > 0 ) {  
 setGlobalEval( destElements, !inPage && getAll( elem, "script" ) );  
 }  
  
 // 返回克隆数据集  
 return clone;  
},

### $().clone------------------(L5976)

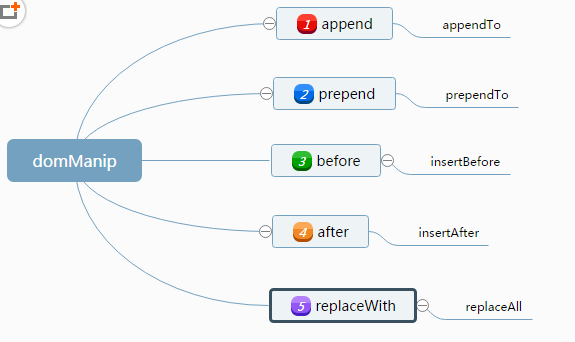
* + - 1. 克隆匹配的元素

/\*\*  
 \* 克隆匹配的DOM元素并且选中这些克隆的副本。  
 \* @param dataAndEvents 是否同时复制元素的附加数据和绑定事件  
 \* @param deepDataAndEvents 是否同时复制元素的所有子元素的附加数据和绑定事件  
 \* @return {\*}  
 \*/  
clone: function( dataAndEvents, deepDataAndEvents ) {  
 dataAndEvents = dataAndEvents == null ? false : dataAndEvents;  
 deepDataAndEvents = deepDataAndEvents == null ? dataAndEvents : deepDataAndEvents;  
  
 return this.map( function() {  
 return jQuery.clone( this, dataAndEvents, deepDataAndEvents );  
 } );  
},

## domManip

### 概述

* + - 1. domManip 的主要功能是为了实现 DOM 的插入和替换，内部后插入（append）、内部前插入（prepend）、外部前插入（before）、外部后插入（after）、替换元素 （replaceWith）
      2. 然后代码中使用each函数，通过上述5个函数生成了appendTo、prependTo、insertBefore、insertAfter、replaceAll



### domManip 的主要功能

* + - 1. 先完成 DOM 节点添加
      2. 如果添加的 DOM 节点内有 script 标签，需要额外处理下。对于可执行的 script （通过type属性判断）则执行其内的脚本代码，其它的则不执行。

### disableScript----------(L5644)

// 更改了script标签的type值以确保安全  
// 原来的type值是"text/javascript"，改成了"true/text/javascript"或"false/text/javascript"  
// 注意：对于script脚本  
// 1.script 无 type 属性，默认会执行其内的 JS 脚本  
// 2.script 的 type="text/javascript" 或 type="text/ecmascript" ，会执行其内的 JS 脚本  
// 3.script 如果有 src 属性，会执行 $.\_evalUrl 请求远程的 JS 文件并执行  
// 4.其它不会执行 JS 脚本，有时我们会用 script 来做 html 模板，  
// 如 underscore.js，type="text/template" 或 type="text/plain" 这种，其内的 JS 都不会被执行  
function disableScript( elem ) {  
 elem.type = ( elem.getAttribute( "type" ) !== null ) + "/" + elem.type;  
 return elem;  
}

### restoreScript----------(L5649)

// disableScipt增加了true，false，这个函数将true和false去除掉  
function restoreScript( elem ) {  
 var match = rscriptTypeMasked.exec( elem.type );  
  
 if ( match ) {  
 elem.type = match[ 1 ];  
 } else {  
 elem.removeAttribute( "type" );  
 }  
 return elem;  
}

### function domManip( collection, args, callback, ignored )-----(L5709)

/\*\*  
 \* domManip 的主要功能是为了实现 DOM 的插入和替换,为append，prepend，before，after，replaceWith使用  
 \* 注意dom操作的几个细节：  
 \* 1、保证最终操作的永远是dom元素，浏览器的最终API只认识那么几个接口  
 \* 2、针对大量操作，需要引入文档碎片做优化  
 \* 浏览器原生的插入节点的方法有两个：appendChild和inserBefore  
 \* @param collection jQuery当前对象集合  
 \* @param args 待插入的DOM元素或HTML代码  
 \* @param callback 回调函数，执行格式为callback.call( 目标元素即上下文, 待插入文档碎片/单个DOM元素 )  
 \* @param ignored 只有在replace调用时，传入了ignored=[]  
 \* @return {\*}  
 \*/

function domManip( collection, args, callback, ignored ) {  
  
 // 将数组和/或值连接成新数组，使任意嵌套数组变平  
 // ([[1,2],[3,4]])---->[1,2,3,4]  
 // ([[1,2,[3]],4])---->[1, 2, Array(1), 4]，并不是展开任意数组  
 args = concat.apply( [], args );  
  
 var fragment, first, scripts, hasScripts, node, doc,  
 i = 0,  
 l = collection.length,  
 iNoClone = l - 1, // 是否为克隆节点，如果当前的jQuery对象所匹配的元素不止一个（n > 1）的话，意味着构建出来的文档碎片需要被n用到，则需要被克隆（n-1）次，加上碎片文档本身才够n次使用  
 value = args[ 0 ], // value 是第一个元素，后边只针对args[0]进行检测，意味着args中的元素必须是统一类型  
 isFunction = jQuery.isFunction( value );// arg第一个参数为函数  
  
 // 在 WebKit中，不能克隆包含了已选中多选按钮的文档碎片,主要是处理Webkit checked属性  
 // 克隆的文档不能重复使用，那么只能是当前jQuery对象所匹配的每个元素都调用一次domManip处理。  
 // 还处理个特殊情况：如果传入的节点是函数（即value是函数）则需要当前jQuery对象所匹配的每个元素都将函数计算出的值作为节点代入domManip中处理  
 // 如传入函数，$('span').append(function (index ,content) {});index为每一个匹配元素，content则是self.html();  
 if ( isFunction ||  
 ( l > 1 && typeof value === "string" &&  
 !support.checkClone && rchecked.test( value ) ) ) {  
 return collection.each( function( index ) {  
 var self = collection.eq( index );  
 if ( isFunction ) { // 如arg第一个参数为函数，则each时将值计算出替换之前函数  
 args[ 0 ] = value.call( this, index, self.html() );  
 }  
 domManip( self, args, callback, ignored );  
 } );  
 }  
 // 对于正常情况，使用传入的节点构建文档碎片  
 // 如$('span').append('<div><h1>asd</h1></div><p>123</p>');  
 // fragment.childNodes.length= 2;  
 // fragment.firstChild为div  
 // 转换HTML代码为DOM元素,  
 if ( l ) {  
 fragment = buildFragment( args, collection[ 0 ].ownerDocument, false, collection, ignored );  
 first = fragment.firstChild;  
  
 if ( fragment.childNodes.length === 1 ) {  
 fragment = first;  
 }  
  
 // 有新内容，即first存在，或ignored为true  
 if ( first || ignored ) {  
 // getAll:根据某个tag名获取context全部tag元素  
 // 原来的type值是"text/javascript"，改成了"true/text/javascript"或"false/text/javascript"  
 scripts = jQuery.map( getAll( fragment, "script" ), disableScript );  
 hasScripts = scripts.length;  
  
 // 将原始文档片段用作最后一个而不是第一个，因为某些情况下，可能错误的被清空(#8070).  
 for ( ; i < l; i++ ) {  
 node = fragment; // 通过buildFragment只构建出一个文档片段  
 // jQuery 是对象集合，都需要这个文档片段，故当i!=iNoClone时，需要克隆这个文档片段  
 if ( i !== iNoClone ) {  
 // 克隆node，同时克隆当前与子元素数据与事件绑定  
 node = jQuery.clone( node, true, true );  
  
 // 保持对克隆脚本的引用，以便以后恢复  
 if ( hasScripts ) {  
  
 // Support: Android <=4.0 only, PhantomJS 1 only  
 // 老版本WebKit调用push.apply(\_, arraylike) 会抛出异常  
 jQuery.merge( scripts, getAll( node, "script" ) );  
 }  
 }  
 // 执行回调函数插入DOM元素  
 callback.call( collection[ i ], node, i );  
 }  
  
 if ( hasScripts ) {  
 // 对于如$('span').append('<div><h1>asd</h1></div><p>123</p>');  
 // 开始时scripts = 1；但如span匹配到4个元素，则collection.length= 4；  
 // 故jQuery.merge( scripts, getAll( node, "script" ) );会运行3次,外面有if( i !== iNoClone )  
 doc = scripts[ scripts.length - 1 ].ownerDocument;  
  
 // 重新启用脚本  
 // 就是将脚本script的true/text/javascript转换为text/javascript  
 jQuery.map( scripts, restoreScript );  
  
 // 在第一个文档插入时，执行脚本  
 for ( i = 0; i < hasScripts; i++ ) {  
 node = scripts[ i ];  
 // 匹配/javascript或ecmascript  
 if ( rscriptType.test( node.type || "" ) &&  
 //dataPriv无数据则存储globalEval  
 !dataPriv.access( node, "globalEval" ) &&  
 jQuery.contains( doc, node ) ) {  
 // 如script包含src属性，则用evalUrl，异步执行  
 if ( node.src ) {  
  
 // 可选的Ajax依赖项，但如果不存在，则不会运行脚本。  
 if ( jQuery.\_evalUrl ) {  
 jQuery.\_evalUrl( node.src );  
 }  
 } else {  
 // 把一段脚本加载到全局context（window）中  
 DOMEval( node.textContent.replace( rcleanScript, "" ), doc );  
 }  
 }  
 }  
 }  
 }  
 }  
  
 return collection;  
}

## append,prepend,before,after------(L5924)

* + - 1. domManip，克隆后的elem并不是jQuery对象，而是普通的element元素

// 向每个匹配的元素内部追加内容。  
append: function() {  
 return domManip( this, arguments, function( elem ) {  
 if ( this.nodeType === 1 || this.nodeType === 11 || this.nodeType === 9 ) {  
 var target = manipulationTarget( this, elem );  
 target.appendChild( elem );// HTML DOM 方法：appendChild() 方法向节点添加最后一个子节点。  
 }  
 } );  
},  
// 向每个匹配的元素内部前置内容。  
prepend: function() {  
 return domManip( this, arguments, function( elem ) {  
 if ( this.nodeType === 1 || this.nodeType === 11 || this.nodeType === 9 ) {  
 var target = manipulationTarget( this, elem );  
 // parentElement.insertBefore(newElement, referenceElement);  
 // 参考节点之前插入一个节点作为一个指定父节点的子节点。  
 target.insertBefore( elem, target.firstChild );  
 }  
 } );  
},  
// 在每个匹配的元素之前插入内容。  
before: function() {  
 return domManip( this, arguments, function( elem ) {  
 if ( this.parentNode ) {  
 this.parentNode.insertBefore( elem, this );  
 }  
 } );  
},  
// 在每个匹配的元素之后插入内容。  
after: function() {  
 return domManip( this, arguments, function( elem ) {  
 if ( this.parentNode ) {  
 // parentDiv.insertBefore(sp1, sp2.nextSibling);  
 // 如果 sp2 没有下一个节点，则它肯定是最后一个节点，  
 // 则 sp2.nextSibling 返回 null，且 sp1 被插入到子节点列表的最后面（即 sp2 后面）。  
 this.parentNode.insertBefore( elem, this.nextSibling );  
 }  
 } );  
},

## replaceWith----(L6024)

### jQuery.cleanData-----------(L5868)

* + - 1. 此函数在jQuery Api并没有，jQuery可能并不希望用户自行删除dataPriv数据
      2. 因此与jQuery.removeData最主要区别是，removeData删除的是dataUser数据

// 删除元素elems绑定的events和userdata  
cleanData: function( elems ) {  
 var data, elem, type,  
 special = jQuery.event.special,  
 i = 0;  
  
 for ( ; ( elem = elems[ i ] ) !== undefined; i++ ) {  
 if ( acceptData( elem ) ) { //判断绑定数据的目标owner类型是否符合要求  
 if ( ( data = elem[ dataPriv.expando ] ) ) {  
 if ( data.events ) {  
 for ( type in data.events ) {  
 if ( special[ type ] ) {  
 jQuery.event.remove( elem, type );  
  
 // 删除事件的简便方法  
 } else {  
 jQuery.removeEvent( elem, type, data.handle );  
 }  
 }  
 }  
  
 // Support: Chrome <=35 - 45+  
 // 设置undefined值，而不是用delete删除属性, see Data#remove  
 elem[ dataPriv.expando ] = undefined;  
 }  
 if ( elem[ dataUser.expando ] ) {// 如有用户数据  
  
 // Support: Chrome <=35 - 45+  
 // 设置undefined值，而不是用delete删除属性, see Data#remove  
 elem[ dataUser.expando ] = undefined;  
 }  
 }  
 }  
}

### function buildFragment( elems, context, scripts, selection, ignored )--(L4766)

* + - 1. 主要针对，ignored参数

/\*\*  
 \* 构建文档片段，jQuery.parseHTML与domManip (主要功能是为了实现 DOM 的插入和替换)调用此函数创建文档片段  
 \* 对于，$('span').replaceWith('<p>123</p>');，传递给domManip( collection, args, callback, ignored )分别为  
 \* collection:为匹配的span的jQuery集合，args为<p>123</p>，callback为replaceWith传递的回调函数，ignored为[]  
 \* 传递给buildFragment则是： elems：<p>123</p>，  
 \* context：document；  
 \* scripts：false,  
 \* selection:为匹配的span的jQuery集合,  
 \* ignored :[],!![]为true  
 \* @param elems  
 \* @param context 上下文  
 \* @param scripts scripts 参数只在 jQuery.parseHTML 方法里使用（domManip里传false），  
 \* 当 jQuery.parseHTML 的第三个参数 keepScripts 为 false 时将删除节点里所有的 script tag  
 \* @param selection  
 \* @param ignored  
 \* @return {DocumentFragment}  
 \*/  
function buildFragment( elems, context, scripts, selection, ignored ) {  
 var elem, tmp, tag, wrap, contains, j,  
 // 在context下创建一个新的空白的文档片段，  
 // 文档片段存在于内存中，并不在DOM树中，所以将子元素插入到文档片段时不会引起页面回流  
 fragment = context.createDocumentFragment(),  
 nodes = [],  
 i = 0,  
 l = elems.length;  
 // parseHTML调用此函数时，elems=[data]，data为外部传入的html，因此仅循环一次  
 // domManip调用elems，是参数，如$('span').append('<h1>asd</h1>');elem为<h1>asd</h1>  
 for ( ; i < l; i++ ) {  
 elem = elems[ i ];  
  
 if ( elem || elem === 0 ) {  
  
 // 因为jQuery.parseHTML限定了data==string，故为false  
 // elem 是 DOM 元素（根据nodeType判断），直接放入 nodes 数组中  
 if ( jQuery.type( elem ) === "object" ) {  
  
 // Support: Android <=4.0 only, PhantomJS 1 only  
 // push.apply(\_, arraylike) 会抛出异常在老的WebKit  
 jQuery.merge( nodes, elem.nodeType ? [ elem ] : elem );  
  
 // 将非HTML转换为文本节点，rhtml= /<|&#?\w+;/  
 // elem 是字符串且不是 HTML tag，创建文本节点对象（textNode），放入 nodes 数组中  
 } else if ( !rhtml.test( elem ) ) {  
 // document.creatTextNode = ‘<h1>rex</h1’>，字符串中具有html代码，不会被浏览器解析，但innerHTML会解析字符串中的html代码，表现为h1标记的rex  
 nodes.push( context.createTextNode( elem ) );  
  
 // 将html转换为dom节点  
 // elem 是字符串且是 HTML tag，将其转成 DOM 元素，放入 nodes 数组中  
 } else {  
 tmp = tmp || fragment.appendChild( context.createElement( "div" ) );  
  
 // 反序列化的标准表示法，  
 // rtagName = ( /<([a-z][^\/\0>\x20\t\r\n\f]+)/i )  
 // [ "", "" ][1]===""获得一个空字符串？？？？？？？？？？？？？？？不知为何这么表示，可能是反序列化标准表示法  
 tag = ( rtagName.exec( elem ) || [ "", "" ] )[ 1 ].toLowerCase();  
 // wrapMap为为了支持xhtml而提供的对节点的匹配，wrapMap.\_default: [ 0, "", "" ]  
 // 例如elem= <tr>rextao</tr>,需要在tr外围增加tbody与table  
 // 如wrapMap.tr = [ 2, "<table><tbody>", "</tbody></table>" ]，wrapMap[0],表示当前tr外围应添加几个元素  
 wrap = wrapMap[ tag ] || wrapMap.\_default;  
 // HTML5不要求标签必须闭合，但是XML要求。这个函数就是用来作转换的。  
 // 如 <h1 id='a'/>是在htmnl5是允许的，通过htmlPreFilter会转换为<h1 id='a'></h1>  
 tmp.innerHTML = wrap[ 1 ] + jQuery.htmlPrefilter( elem ) + wrap[ 2 ];  
   
 // 对于$.parseHTML调用此函数，通过j--，tmp得到的还是如上举例的tr元素，并不能看出有何用  
 j = wrap[ 0 ];  
 while ( j-- ) {  
 tmp = tmp.lastChild;  
 }  
  
 // Support: Android <=4.0 only, PhantomJS 1 only  
 // push.apply(\_, arraylike) 会抛出异常在老的WebKit  
 jQuery.merge( nodes, tmp.childNodes );  
  
 // 记住顶级容器，即上面创建的div  
 tmp = fragment.firstChild;  
  
 // 确保创建的节点是孤立的 (bug#12392)  
 tmp.textContent = "";  
 }  
 }  
 }  
  
 // 移除fragment内部所有的内容  
 fragment.textContent = "";  
  
 i = 0;  
 while ( ( elem = nodes[ i++ ] ) ) {  
  
 // 跳过contex集合中包含elements的情况(trac-4087)，$.parseHTML调用不涉及  
 if ( selection && jQuery.inArray( elem, selection ) > -1 ) {  
 // 何时才会push呢？如replaceWith这样调用：$('p').replaceWith($('#p1'));那么elem在selection集合中  
 // 故会将ignored集合push个elem元素  
 if ( ignored ) {  
 ignored.push( elem );  
 }  
 continue;  
 }  
 // jQuery.contains = Sizzle.contains,判断一个DOM节点是否包含另一个DOM节点  
 contains = jQuery.contains( elem.ownerDocument, elem );  
  
 // 添加到 fragment,getAll(context,tag):根据某个tag名获取context全部tag元素  
 tmp = getAll( fragment.appendChild( elem ), "script" );  
  
 // 保存脚本  
 if ( contains ) {  
 setGlobalEval( tmp );  
 }  
  
 // 获取脚本，如scripts为true，此处将脚本存入scripts中，因为scripts为true，  
 // $.parseHTML中会remove scripts  
 if ( scripts ) {  
 j = 0;  
 while ( ( elem = tmp[ j++ ] ) ) {  
 // rscriptType = ( /^$|\/(?:java|ecma)script/i )  
 // 匹配script脚本，匹配的是type， script的elem.type=text/javascript  
 if ( rscriptType.test( elem.type || "" ) ) {  
 scripts.push( elem );  
 }  
 }  
 }  
 }  
  
 return fragment;  
}

### replaceWith------(L6024)

// 将所有匹配的元素替换成指定的HTML或DOM元素。  
replaceWith: function() {  
 var ignored = []; // 因为dooManip中循环调用此func函数，buildFragment会根据情况往ignored中push函数  
  
 // 用新内容替换每个未忽略的上下文元素  
 return domManip( this, arguments, function( elem ) {  
 var parent = this.parentNode;  
 // 查看this是否在ignored数组中  
 if ( jQuery.inArray( this, ignored ) < 0 ) {  
 // 清空this上的dataPriv与dataUser  
 jQuery.cleanData( getAll( this ) );  
 if ( parent ) {  
 // 用指定的节点替换当前节点的一个子节点，并返回被替换掉的节点  
 parent.replaceChild( elem, this );  
 }  
 }  
  
 // 强制调用callback  
 }, ignored );  
}

## appendTo, prependTo, insertBefore等-----(L6043)

// 与append，prepend等重要不同是，是颠倒了常规的$(A).append(B)的操作，即不是把B追加到A中，而是把A追加到B中。  
// 方法就是用each，将$(A).append(B)，将A，B参数进行交换而已  
jQuery.each( {  
 appendTo: "append", // $("<b>Hello World!</b>").appendTo("p");在每个 p 元素结尾插入内容：  
 prependTo: "prepend",  
 insertBefore: "before",  
 insertAfter: "after",  
 replaceAll: "replaceWith"  
}, function( name, original ) {  
 jQuery.fn[ name ] = function( selector ) {  
 var elems,  
 ret = [],  
 insert = jQuery( selector ),  
 last = insert.length - 1,  
 i = 0;  
  
 for ( ; i <= last; i++ ) {  
 // 如insert有多个的时候就需要完全克隆一份副本  
 elems = i === last ? this : this.clone( true );  
 jQuery( insert[ i ] )[ original ]( elems );  
  
 // Support: Android <=4.0 only, PhantomJS 1 only  
 // 使用.get()是因为push.apply(\_, arraylike)会在老的WebKit抛出异常  
 push.apply( ret, elems.get() );  
 }  
  
 return this.pushStack( ret );  
 };  
} );

## 从DOM中删除所有匹配的元素

### 对外接口$().remove();$().detach();---------------(L5904)

// 从DOM中删除所有匹配的元素。  
// 这个方法不会把匹配的元素从jQuery对象中删除，  
// 因而可以在将来再使用这些匹配的元素。与remove()不同的是，所有绑定的事件、附加的数据等都会保留下来。  
detach: function( selector ) {  
 return remove( this, selector, true );  
},  
// 从DOM中删除所有匹配的元素。  
// 所有绑定的事件、附加的数据等不会保留  
// 对于$().remove()操作，会删除当前元素  
remove: function( selector ) {  
 return remove( this, selector );  
},

### 内部函数function remove( elem, selector, keepData ) ----------(L5799)

/\*\*  
 \* remove，detach的内部删除函数  
 \* @param elem  
 \* @param selector  
 \* @param keepData 是否保留节点数据，true为保留  
 \* @return {\*}  
 \*/  
function remove( elem, selector, keepData ) {  
 var node,  
 // jQuery.filter:筛选出与指定表达式匹配的元素集合  
 // 因此如果selector为null，则nodes为elem，会删除当前节点  
 nodes = selector ? jQuery.filter( selector, elem ) : elem,  
 i = 0;  
  
 for ( ; ( node = nodes[ i ] ) != null; i++ ) {  
 if ( !keepData && node.nodeType === 1 ) {  
 jQuery.cleanData( getAll( node ) );  
 }  
  
 if ( node.parentNode ) {  
 if ( keepData && jQuery.contains( node.ownerDocument, node ) ) {  
 // 标记script是否已经执行过  
 setGlobalEval( getAll( node, "script" ) );  
 }  
 node.parentNode.removeChild( node );  
 }  
 }  
  
 return elem;  
}

## $().empty-------(L5958)

* + - 1. 主要是利用elem.textContent，设置属性会将elem子元素全部删除
      2. 删除匹配的元素集合中所有的子节点。

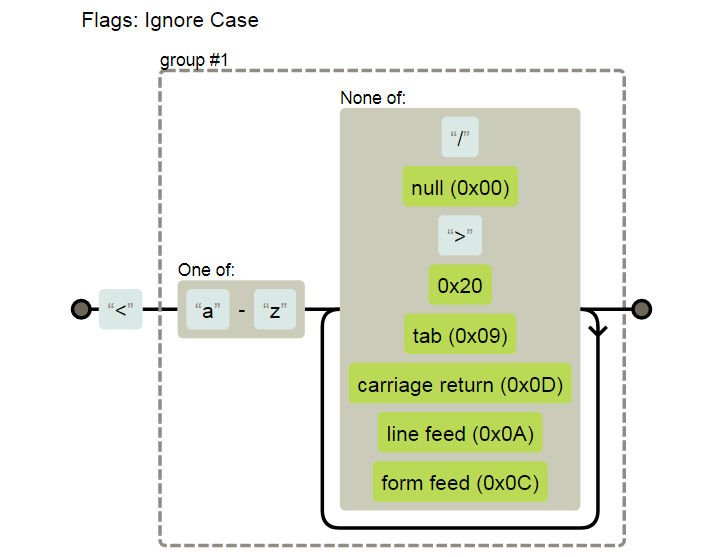
// 删除匹配的元素集合中所有的子节点。  
empty: function() {  
 var elem,  
 i = 0;  
  
 for ( ; ( elem = this[ i ] ) != null; i++ ) {  
 if ( elem.nodeType === 1 ) {  
  
 // 防止内存泄漏  
 jQuery.cleanData( getAll( elem, false ) );  
  
 // 删除所有剩余节点  
 // 一个节点及其后代的文本内容  
 // 在节点上设置 textContent 属性的话，会删除它的所有子节点  
 elem.textContent = "";  
 }  
 }  
  
 return this;  
},

## $().html()----------------

### rtagName-----(L4695)

var rtagName = ( /<([a-z][^\/\0>\x20\t\r\n\f]+)/i );

* + - 1. 匹配类似<a123



### $().html() ---------------------(L5985)

// 获取集合中第一个匹配元素的HTML内容

html: function( value ) {  
 // 获取或设置一个集合的值,value如是函数，可以选择是否执行  
 return access( this, function( value ) {  
 var elem = this[ 0 ] || {},  
 i = 0,  
 l = this.length;  
 // $().html()调用方式，返回elem的innerHTML  
 // innerHTML:属性设置或获取HTML语法表示的元素的后代。  
 if ( value === undefined && elem.nodeType === 1 ) {  
 return elem.innerHTML;  
 }  
  
 // 看是否能直接使用innerHTML  
 // rnoInnerhtml:需要执行的代码script|style|link等不能使用innerHTML  
 // wrapMap:必须关闭这些标签来支持XHTML  
 if ( typeof value === "string" && !rnoInnerhtml.test( value ) &&  
 !wrapMap[ ( rtagName.exec( value ) || [ "", "" ] )[ 1 ].toLowerCase() ] ) {  
 // 将单标签的进行转换，HTML5不要求标签必须闭合，但是XML要求  
 value = jQuery.htmlPrefilter( value );  
  
 try {  
 for ( ; i < l; i++ ) {  
 elem = this[ i ] || {};  
  
 // 删除元素节点并防止内存泄漏。  
 if ( elem.nodeType === 1 ) {  
 jQuery.cleanData( getAll( elem, false ) );  
 elem.innerHTML = value;  
 }  
 }  
  
 elem = 0;  
  
 // 如用innerHTML抛出异常，则使用回退方法  
 } catch ( e ) {}  
 }  
  
 if ( elem ) {  
 // 如innerHTML报错，则用empty清空，然后用append附加上value  
 this.empty().append( value );  
 }  
 }, null, value, arguments.length );  
},

### $().text() -----------------(L5912)

* + - 1. text与html内部实现重要不同是，text使用的是textContent，html使用的innerHTML

// $().text():返回匹配元素文本的拼接字符串  
// $().text('');：设置文本  
text: function( value ) {  
 return access( this, function( value ) {  
 return value === undefined ?  
 // jQuery.text = Sizzle.getText;/ 用于检索DOM节点数组的文本值的工具函数  
 // Sizzle.getText：返回的是拼接的字符串  
 jQuery.text( this ) :  
 this.empty().each( function() {  
 if ( this.nodeType === 1 || this.nodeType === 11 || this.nodeType === 9 ) {  
 this.textContent = value;  
 }  
 } );  
 }, null, value, arguments.length );  
},

## 包裹wrapAll,wrap,wrapInner,unwrap

### $().wrapAll()------------------(L9350)

// 将所有匹配的元素用单个元素包裹起来  
// 注意，因为$()获取的元素在dom树中，buildFrament利用原生的elem.appendChild(),这个方法会先删除原dom树的节点，然后再添加  
wrapAll: function( html ) {  
 var wrap;  
  
 if ( this[ 0 ] ) {  
 if ( jQuery.isFunction( html ) ) {  
 html = html.call( this[ 0 ] );  
 }  
  
 // 获得包裹标签  
 // jQuery(html,document); 根据html创建临时dom  
 wrap = jQuery( html, this[ 0 ].ownerDocument ).eq( 0 ).clone( true );  
  
 if ( this[ 0 ].parentNode ) {// 如this[ 0 ]存在parent，则将wrap插在this[ 0 ]之前  
 wrap.insertBefore( this[ 0 ] );  
 }  
 // map: function( callback ) {  
 // return this.pushStack( jQuery.map( this, function( elem, i ) {  
 // return callback.call( elem, i, elem );  
 // } ) );  
 // }  
 // wrap.map返回的是this.pushStack(value)  
 // value 则是jQuery.map( this，func)值  
 // jQuery.map中重要的一步是,value = callback( elems[ i ], i, arg );此处callback就是wrap.map(func)的func  
 wrap.map( function() {  
 var elem = this;  
 // 返回对象的第一个孩子 Element  
 while ( elem.firstElementChild ) {  
 elem = elem.firstElementChild;  
 }  
  
 return elem;  
 } ).append( this );  
 }  
  
 return this;  
},

### $().wrapInner-----------------------(L9379)

// 将每一个匹配的元素的子内容(包括文本节点)用一个HTML结构包裹起来  
wrapInner: function( html ) {  
 if ( jQuery.isFunction( html ) ) {  
 return this.each( function( i ) {  
 jQuery( this ).wrapInner( html.call( this, i ) );  
 } );  
 }  
  
 return this.each( function() {  
 var self = jQuery( this ),  
 contents = self.contents(); //查找匹配元素内部所有的子节点（包括文本节点）  
 // 对于$('li'),通过self.contents(),获取匹配li的子节点，然后在wrapAll，就是将子节点包裹了html  
 if ( contents.length ) {  
 contents.wrapAll( html );  
 // 如节点内无内容，就是往节点内append，html  
 } else {  
 self.append( html );  
 }  
 } );  
},

### $().wrap-----------------------(L9399)

// 把所有匹配的元素用其他元素的结构化标记包裹起来。  
// 其实是将每个匹配元素，调用wrapAll包裹  
wrap: function( html ) {  
 var isFunction = jQuery.isFunction( html );  
  
 return this.each( function( i ) {  
 jQuery( this ).wrapAll( isFunction ? html.call( this, i ) : html );  
 } );  
},

### Wrap,wrapAll,wrapInner区别

* + - 1. 原始HTML

<ul>  
 <li title='苹果'>苹果</li>  
 <li title='橘子'>橘子</li>  
 <li title='菠萝'>菠萝</li>  
</ul>

* + - 1. $("li").wrap("<div></div>");每一个选择器都添加

<ul>  
 <div><li title="苹果">苹果</li></div>  
 <div><li title="橘子">橘子</li></div>  
 <div><li title="菠萝">菠萝</li></div>  
</ul>

* + - 1. $("li").wrapAll("<div></div>");在所有选中的选择器最外面添加

<ul>  
 <div>  
 <li title="苹果">苹果</li>  
 <li title="橘子">橘子</li>  
 <li title="菠萝">菠萝</li>  
 </div>  
</ul>

* + - 1. $("li").wrapInner("<div></div>");为选择器的内容添加

<ul>  
 <li title='苹果'><div>苹果</div></li>  
 <li title='橘子'>><div>橘子</div></li>  
 <li title='菠萝'>><div>菠萝</div></li>  
</ul>

* + - 1. 参考：http://blog.sina.com.cn/s/blog\_780a942701015nac.html

### $().unwrap()------------------------(L9407)

// 移除每个匹配元素的父元素  
unwrap: function( selector ) {  
 this.parent( selector ).not( "body" ).each( function() {  
 // 用匹配元素父节点的所有子节点替换匹配元素的父节点  
 // 当然了父节点是body/html/document肯定是移除不了的  
 jQuery( this ).replaceWith( this.childNodes );  
 } );  
 return this;  
}