# 整体架构-jQuery3.2.1

## 整体架构总览

( function( global, factory ) {

…………………….

} )( typeof window !== "undefined" ? window : this, function( window, noGlobal ){

var  
 version = "3.2.1",  
 jQuery = function( selector, context ) {  
 return new jQuery.fn.init( selector, context );  
 }

jQuery.fn = jQuery.prototype = {}

// (114-194)给JQ原型对象，添加一些方法和属性

// (195-263) extend : JQ的extend函数

// (264-537) jQuery.extend() : 扩展一些工具函数

// (538-2807):选择器Sizzle引擎

// (2808-3241):dom元素筛选方法和初始化jQuery对象(2947-3064)

// (3242-3467):回调函数jQuery.Callbacks

// (3468-3863):异步队列jQuery.Deferred

// (3864-3951)：jQuery.ready实现

// (3952-4352):数据缓存$.data();

// (4353-4486):队列操作,queue,dequeue,promise()

// (4693-4881):DOM操作0，DOM操作内部函数buildFragment等

// (4882-5612):事件系统1

// (5613-6068):DOM操作1，append，prepend等

// (6069-6652)：样式操作：$().css();

// (7461-7614)：属性操作1：$().attr();$().removeAttr();

// (7615-7752)：属性操作2：$().prop();$().removeProp();

// (7753-7923)：Class操作：$().addClass(), $().removeClass(),$().hasClass(),$().toggleClass();

// (7924-8108)：属性操作3：$().val();

// (8109-8347):事件系统2, trigger, triggerHandler

// (9348-9414): DOM操作2,wrap,wrapAll,unWrap

// (9911-10119) :位置操作，postion，offset，scrollLeft等

// (10120-10165) :尺寸操作，innerHeight, innerWidth, height, width, outerHeight and outerWidth这几个方法

// (10166-10185) :事件系统3, bind,unbind, delegate,undelegate

//

return jQuery;

})

## 立即调用表达式

( function( global, factory ) {

…………………….

} )( typeof window !== "undefined" ? window : this, function( window, noGlobal ){})

* + - 1. 其中global，factory是自调用函数的形参
      2. Line3，表示为自调用函数传入2个参数，一个是window或this，第二个参数为一个函数
      3. 如window为undefined，说明使用jQuery环境不存在window对象，故传入this作为global，如存在window对象，则global为window
      4. Window对象传入，主要是为了减少变量查找所经过的scope作用域
      5. jQuery利用了JavaScript函数作用域的特性，采用立即调用表达式包裹了自身的方法来解决命名空间与变量污染的问题。

## 类数组

* + - 1. jQuery主要是为了方便DOM操作，为了让节点与实例对象通过一个桥梁给关联起来，内部就采用了一种叫“类数组对象”的方式作为存储结构，所以我们可以像对象一样，也可像数组一样操作jQuery对象
      2. 给出如何通过#book，直接获取到book的dom

document.getElementById(/[^#].\*/.exec(selector)[0]);

* + - 1. 因为代码中this=elem，而且还提供了get方法，故可以用数组和对象两种方式访问

var aQuery = function(selector) {  
 //强制为对象  
 if (!(this instanceof aQuery)) {  
 return new aQuery(selector);  
 }  
 var elem = document.getElementById(/[^#].\*/.exec(selector)[0]);  
 this.length = 1;  
 this[0] = elem;  
 this.context = document;  
 this.selector = selector;  
 this.get = function(num) {  
 return this[num];  
 };  
 return this;  
};  
$("#test3").click(function() {  
 $('#show3').append(aQuery("#book")[0])  
});  
$("#test4").click(function() {  
 $('#show4').append(aQuery("#book").get(0))  
});

* + - 1. 但是每次new一个对象会创建了一个新的实例，那么类似get方法就要在每一个实例上重新创建一遍，性能就大打折扣

## 对象的构建

* + - 1. 类一

function AjQuery() {  
 this.name = 'jQuery';  
 this.sayName = function () {  
 return this.name  
 };  
}  
var a = new AjQuery();  
var b = new AjQuery();  
var c = new AjQuery();

* + - 1. 类二

function AjQuery() {  
 this.name = 'jQuery'  
}  
AjQuery.prototype = {  
 sayName: function() {  
 return this.name  
 }  
};  
var a = new AjQuery();  
var b = new AjQuery();  
var c = new AjQuery();

* + - * 1. 类一：会为每个实例复制sayName方法，如果把所有属性方法都声明在构造函数中，就会无形的增大很多开销；
        2. 类二：new的几个实例共享sayName方法，通过原型链查找找到sayName方法，多了一层原型链查找
      1. New操作符构建一个对象
         1. 创建一个新对象
         2. 将构造函数的作用域赋给新对象（所以this就指向了这个新对象）
         3. 执行构造函数中的代码
         4. 返回这个新对象

**new操作符主要是把原型链跟实例的this关联起来，这才是最关键的一点**

## jQuery的无new构建

（1）jQuery没有使用new运行符将jQuery显示的实例化，是直接调用其函数，故jQuery不是像如下方式实现

var jQuery = function(selector, context) {  
 //构造函数  
};  
jQuery.prototype = {  
 //原型  
 name:function(){},  
 age:function(){}  
};  
var a = new jQuery();  
a.name();

（2）$().ready()和$().noConflict()类似调用，故jQuery应当看做一个类，返回类的实例才对

var aQuery = function(selector, context) {  
 return new aQuery();  
}  
aQuery.prototype = {  
 name:function(){},  
 age:function(){}  
}

通过new aQuery()，但明显是死循环，每次调用aQuery()又会new。。。

（3）返回正确实例——**javascript中实例this只跟原型有关系**

var aQuery = function(selector, context) {  
 return new aQuery.prototype.init();  
};  
aQuery.prototype = {  
 init:function(){  
 return this;  
 },  
 name:function(){},  
 age:function(){}  
};  
console.log(aQuery());

init中的this指向aQuery类的实例，但当把init函数当做构造器时，就会出问题；如下，本应结果为20的，却得到18

var aQuery = function(selector, context) {  
 return aQuery.prototype.init();  
};  
aQuery.prototype = {  
 init: function() {

console.log(this)//20  
 this.age = 18

console.log(this)/18  
 return this;  
 },  
 name: function() {},  
 age: 20  
};  
console.log(aQuery().age );//18

（4）jQuery框架分隔作用域的处理

var aQuery = function(selector, context) {  
 return new aQuery.prototype.init();  
};  
aQuery.prototype = {  
 init: function() {  
 this.age = 18;  
 console.log(this);//{age:18}  
 return this;  
 },  
 name: function() {},  
 age: 20  
};  
//Uncaught TypeError: Object [object Object] has no method 'name'  
console.log(aQuery().name())

通过实例init函数，每次都构建新的init实例对象，来分隔this，避免交互混淆；**但是init中的this不再指向aQuery，而是指向new出来的新实例**

（5）如何访问到jQuery原型上的属性和方法

通过原型传递解决问题，即把jQuery的原型传递给，init原型上，jQuery的原型对象覆盖了init构造器的原型对象，因为是引用传递所以不需要担心这个循环引用的性能问题

var aQuery = function(selector, context) {  
 return new aQuery.prototype.init();  
};  
aQuery.prototype = {  
 init: function() {  
 return this;  
 },  
 name: function() {return this.age},  
 age: 20  
};  
aQuery.prototype.init.prototype = aQuery.prototype;  
console.log(aQuery().name())//20

源码：line114，故可以fn表示prototype

jQuery.fn = jQuery.prototype

## 引用传递与值传递

（1）按值传递 VS. 按引用传递

值传递：函数的形参是被调用时所传实参的副本。修改形参的值并不会影响实参

引用传递：函数形参的值如果被修改，实参也会被修改

（2）JS的基本类型，是按值传递

var a = 1;  
function foo(x) {  
 x = 2;  
}  
foo(a);  
console.log(a); // 仍为1, 未受x = 2赋值所影响

（3）js对象

var obj = {x : 1};  
function foo(o) {  
 o.x = 3;  
}  
foo(obj);  
console.log(obj.x); // 3, 被修改了!

var obj = {x : 1};  
function foo(o) {  
 o = 3;  
}  
foo(obj);  
console.log(obj.x); // 1, 并未修改

js对象传递既不是引用传递也不是值传递

（3）共享传递

调用函数传参时，函数接受对象实参**引用的副本**(既不是按值传递的对象副本，也不是按引用传递的隐式引用)

与引用传递的不同在于：在共享传递中对函数形参的赋值，不会影响实参的值；但形参可以为obj修改或添加属性值

（4）基本类型的不可变

var str = "abc";  
console.log(str[0])// a  
str[0] = 1;  
console.log(str); // abc, 并未修改

string对象是不可变的，并不能改变字符串内容，js修改字符串，实际是创建新的string值

## 链式调用

1. DOM链式调用的处理

——节约JS代码

——所返回的都是同一个对象，可以提高代码的效率

（2）**实现链式的基本条件就是实例this的存在，并且是同一个**

## 插件接口

（1）目的

为开发者提供一个扩展jQuery与jQuery.prototype的接口，字面上不是直接修改prototype

（2）Line195，**jQuery.extend和jQuery.fn.extend其实是同指向同一方法的不同引用**

jQuery.extend = jQuery.fn.extend = function() {

这个连等表明指向同一个函数，如何实现不同功能呢？？？主要是this力量

jQuery.extend //对jQuery本身的属性和方法进行了扩展  
jQuery.fn.extend // 对jQuery.fn的属性和方法进行了扩展

* jQuery.extend 调用的时候，this是指向jQuery对象 (jQuery是函数，也是对象！)，所以这里扩展在jQuery上。
* jQuery.fn.extend 调用的时候，this指向fn对象，jQuery.fn 和jQuery.prototype指向同一对象，扩展fn就是扩展jQuery.prototype原型对象。

## ready与load事件

* + - 1. 执行顺序

——解析HTML结构。

——加载外部脚本和样式表文件。

——解析并执行脚本代码。

——构造HTML DOM模型。//ready

——加载图片等外部文件。

——页面加载完毕。//load

* + - 1. ？？？？？？？？？？？？？？？？？？？？jQuery的ready，onload机制

## 多库并存处理

* + - 1. 总会遇到两种情况
         1. $可能与其他库发生冲突
         2. jQuery版本更新太快，插件跟不上，导致不同版本对插件的支持度不一样。
      2. noConflict函数使用方法

jQuery.noConflict();  
// 使用 jQuery  
jQuery("aaron").show();  
// 使用其他库的 $()  
$("aaron").style.display ="book";

* + - 1. Line 10230
         1. 假设一个库，使用了$和jQuery，最后再引入jQuery.js会将原来的$和jQuery覆盖，故可利用noConflict，将jQuery的$和jQuery交出
         2. 除了可以避免第三方库冲突，还可以引用不同版本的jQuery
         3. 在运行noConflict时，因为加载了jQuery.js，window.$ = window.jQuery

——您好，我看您15年时，询问过，jQuery源码中，noConflict实现，为何要加if（window.$ === jQuery），您理解了？我也认为是恒为真？？？？？？？？？？？？？？？？？？？

var  
 // 将window下其他jQuery和$缓存  
 \_jQuery = window.jQuery,   
 \_$ = window.$;  
jQuery.noConflict = function( deep ) {

//此时引入jQuery，故相等，  
 if ( window.$ === jQuery ) {  
 window.$ = \_$;  
 }

//deep为true，jQuery交出jQ-  
 if ( deep && window.jQuery === jQuery ) {  
 window.jQuery = \_jQuery;  
 }  
 return jQuery;  
};

# 创建jQuery对象----(L94)

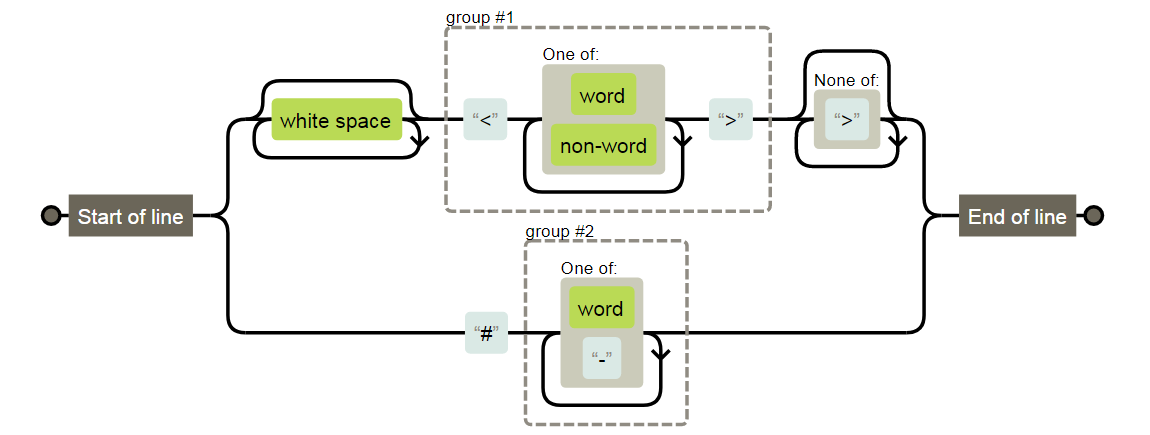
## 局部jQuery变量

// 定义个局部变量jquery，拷贝的是jquery对象  
jQuery = function( selector, context ) {  
  
 // jQuery对象实际上仅仅是init构造函数的提升  
 // 如jQuery被调用则需要初始化  
 return new jQuery.fn.init( selector, context );  
}

## jQuery.fn.init------(L2960)

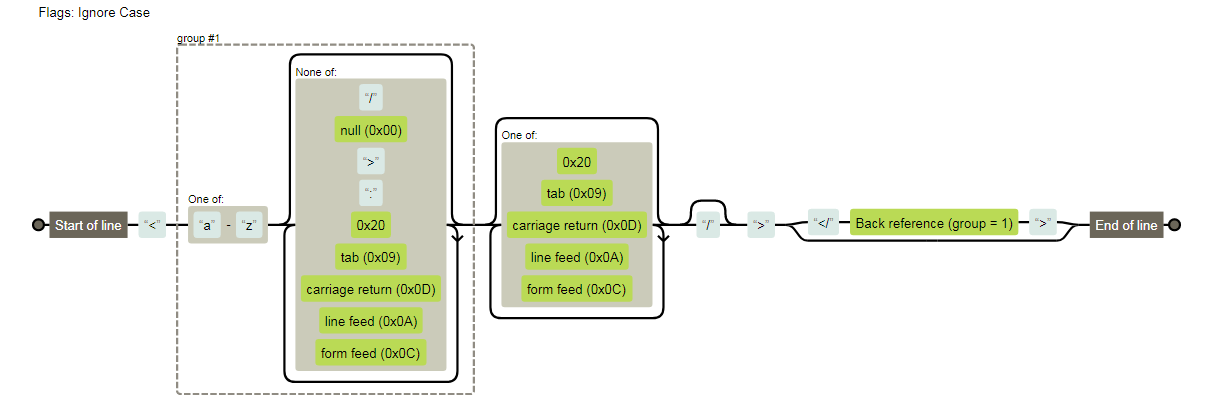
### rquickExpr

* + - 1. 正则匹配id与html标记
      2. rquickExpr = /^(?:#([\w-]+)|(\w+)|\.([\w-]+))$/



### jQuery.parseHTML------(L9771)

* + - 1. rsingleTag = ( /^<([a-z][^\/\0>:\x20\t\r\n\f]\*)[\x20\t\r\n\f]\*\/?>(?:<\/\1>|)$/i )
      2. 此正则匹配单标签如<h1>或<h1></h1>，不能匹配<h1>aa</h1>
      3. (?:<\/\1>|)$/i )表示，匹配</....>或没有，最后一个|很关键，\1表示正则第一个括号匹配的内容反向引用

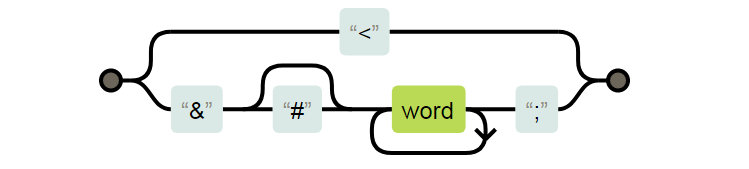


* + - 1. 源码逐行分析（先看2.2.3较好）

/\*\*  
 \* 将字符串解析到一个DOM节点的数组中  
 \* @param data 字符串类型的html  
 \* @param context 如配置，frament会在此context下创建，context默认为document  
 \* @param keepScripts 指定传入的HTML字符串中是否包含脚本，默认为false,如为true，表明在传递的HTML字符串中包含脚本。  
 \* @return {\*} 函数的返回值为Array类型，返回解析指定HTML字符串后的DOM节点数组。  
 \*/  
jQuery.parseHTML = function( data, context, keepScripts ) {  
 if ( typeof data !== "string" ) {  
 return [];  
 }  
 // 如$.parseHTML('',true)形式调用  
 // 等价于，$('',false,true)  
 if ( typeof context === "boolean" ) {  
 keepScripts = context;  
 context = false;  
 }  
  
 var base, parsed, scripts;  
 // 如context为false，即未指定context或$('',boolean)  
 if ( !context ) {  
  
 // Stop scripts or inline event handlers from being executed immediately  
 // by using document.implementation  
 if ( support.createHTMLDocument ) {  
 context = document.implementation.createHTMLDocument( "" );  
  
 // Set the base href for the created document  
 // so any parsed elements with URLs  
 // are based on the document's URL (gh-2965)  
 base = context.createElement( "base" );  
 base.href = document.location.href;  
 context.head.appendChild( base );  
 } else {  
 context = document;  
 }  
 }  
 // ( /^<([a-z][^\/\0>:\x20\t\r\n\f]\*)[\x20\t\r\n\f]\*\/?>(?:<\/\1>|)$/i );  
 // 此正则匹配单标签如<h1>或<h1></h1>，不能匹配<h1>aa</h1>  
 // (?:<\/\1>|)$/i )表示，匹配</....>或没有，最后一个|很关键，\1表示正则第一个括号匹配的内容反向引用  
 // pased[1] =[a-z][^\/\0>:\x20\t\r\n\f]\*  
 parsed = rsingleTag.exec( data );  
 scripts = !keepScripts && [];// !![]为真  
  
 // Single tag单个标签情况  
 if ( parsed ) {  
 // 如<h1>或<h1><h1/>匹配到，则parsed[1],则为h1,即一个dom节点的数组  
 return [ context.createElement( parsed[ 1 ] ) ];  
 }  
 // 把传入的复杂的html转为文档碎片并且存储在jQuery.fragments这个对象里  
 parsed = buildFragment( [ data ], context, scripts );  
  
 if ( scripts && scripts.length ) {//删除其中html中的js脚本  
 // 调用的是，remove: function( selector ) {  
 // return remove( this, selector );  
 // },  
 // 因jQuery.remove()较为复杂，之后再分析  
 // 此处就是利用jQuery(scirpts)找到此脚本，然后删除  
 jQuery( scripts ).remove();  
 }  
 // parsed为frament片段，因此返回childNodes，即是，获得DOM节点数组  
 return jQuery.merge( [], parsed.childNodes );  
};

### buildFragment-----(L4766)

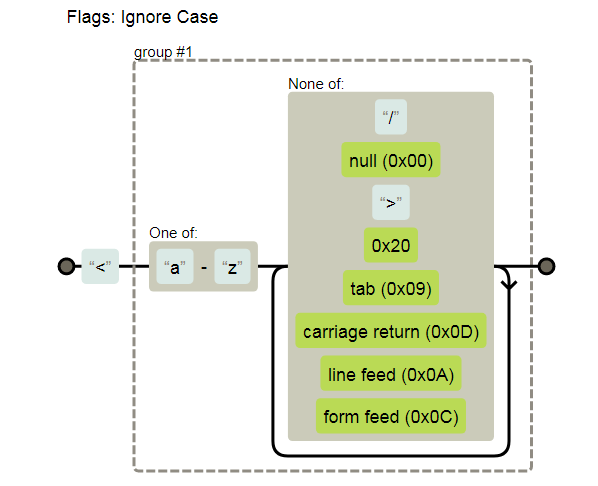
* + - 1. 创建文档片段，jQuery.parseHTML与domManip (主要功能是为了实现 DOM 的插入和替换)调用此函数创建文档片段
      2. rhtml = /<|&#?\w+;/，将非HTML转换为文本节点



* + - 1. jQuery.parseHTML是parsed = buildFragment( [ data ], context, scripts )调用此函数的，故参数selection, ignored并未使用
      2. createTextNode与innerHTML的区别

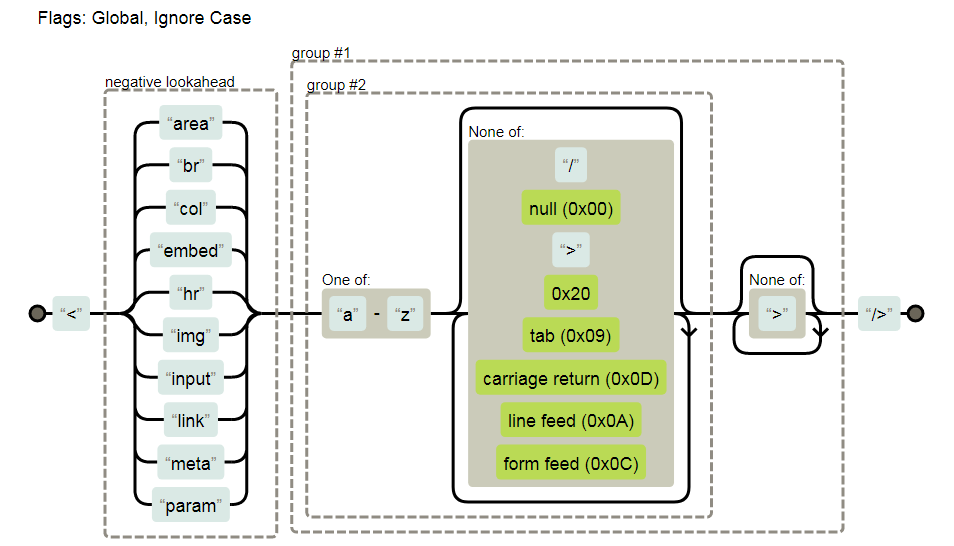
document.creatTextNode = ‘<h1>rex</h1’>，字符串中具有html代码，不会被浏览器解析，但innerHTML会解析字符串中的html代码，表现为h1标记的rex

* + - 1. rtagName = ( /<([a-z][^\/\0>\x20\t\r\n\f]+)/i )，匹配html的在左边标记，即如<h1>asdf</h1>，即匹配出h1



#### htmlPrefilter---(L5821)

* + - 1. rxhtmlTag = /<(?!area|br|col|embed|hr|img|input|link|meta|param)(([a-z][^\/\0>\x20\t\r\ n\f]\*)[^>]\*)\/>/gi,



* + - 1. 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。这样我们使用 .html() 、 .append() 、.replaceWith() 时就不需要人工转换了。因此要忽略单标记的元素如br、area等，即如”<h1 id=’a’/>”.repalce(rxhtmlTag, "<$1></$2>");会返回<h1 id=’a’></h1>

htmlPrefilter: function( html ) {  
 return html.replace( rxhtmlTag, "<$1></$2>" );  
}

* + - 1. wrapMap作用

wrapMap为为了支持xhtml而提供的对节点的匹配，wrapMap.\_default: [ 0, "", "" ]，默认情况下外围不添加，即只是添加””；

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

#### function getAll( context, tag )-----(L4725)

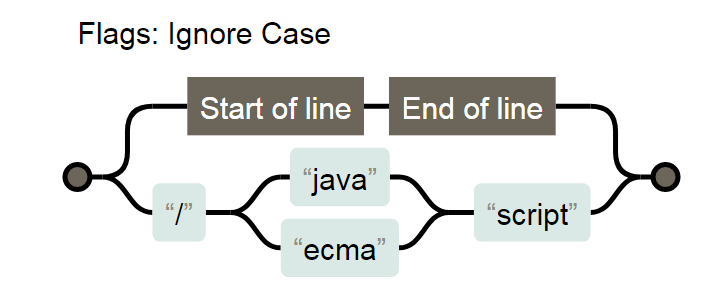
/\*\*  
 \* 根据某个tag名获取context全部tag元素  
 \* @param context  
 \* @param tag  
 \* @return {\*}  
 \*/  
function getAll( context, tag ) {  
  
 // Support: IE <=9 - 11 only  
 // 使用typeof避免对象无参数的方法调用(#15151)  
 var ret;  
  
 if ( typeof context.getElementsByTagName !== "undefined" ) {  
 ret = context.getElementsByTagName( tag || "\*" );  
  
 } else if ( typeof context.querySelectorAll !== "undefined" ) {  
 ret = context.querySelectorAll( tag || "\*" );  
  
 } else {  
 ret = [];  
 }  
 // nodeName:判断elem的nodeName是否为name  
 if ( tag === undefined || tag && nodeName( context, tag ) ) {  
 return jQuery.merge( [ context ], ret );  
 }  
  
 return ret;  
}

* + - 1. function nodeName( elem, name )----(L2842)

// 判断elem的nodeName是否为name  
function nodeName( elem, name ) {  
  
 return elem.nodeName && elem.nodeName.toLowerCase() === name.toLowerCase();  
  
};

* + - 1. rscriptType = ( /^$|\/(?:java|ecma)script/i )

匹配script脚本，匹配的是type类型， script的elem.type=text/javascript



* + - 1. 对于domManip调用在11章，dom操作中

#### 源码逐行分析

/\*\*  
 \* 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  
 \* @param elems  
 \* @param context  
 \* @param scripts  
 \* @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  
 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 ) {  
 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;  
}

### init源码逐行解析

init = jQuery.fn.init = function( selector, context, root ) {  
 var match, elem;  
 // HANDLE: $(""), $(null), $(undefined), $(false)，增加程序的健壮性  
 if ( !selector ) { //将 selector 转成布尔值,如是上述值，jQuery不做任何操作  
 return this;  
 }  
  
 // 默认root为document,但可以传入其他root，主要是用来支持jQuery.sub  
 root = root || rootjQuery;  
  
 // 如selector为string  
 if ( typeof selector === "string" ) {  
 // 处理$('<rextao>')等情况  
 if ( selector[ 0 ] === "<" &&  
 selector[ selector.length - 1 ] === ">" &&  
 selector.length >= 3 ) {//开头结尾分别为<>，并且<>中间至少有一个字符  
 //如是html标签组，不需要正则  
 match = [ null, selector, null ];  
  
 } else {// 匹配$('#rex')或  
 // $('<rextao></asdf>')，总之以<开头以>结尾即可  
 // 根据rquickExpr可知，第一种情况(group1)match[1]=undefined,第二种情况(group2)match[2]=undefined  
 match = rquickExpr.exec( selector );  
 }  
  
 //匹配到html或id（未指定上下文环境）  
 if ( match && ( match[ 1 ] || !context ) ) {  
  
 // HANDLE: $(html) -> $(array),match = rquickExpr.exec( "#id" ),mathc[1]=undefined  
 if ( match[ 1 ] ) {//匹配类似$('<rextao></asdf>')  
 context = context instanceof jQuery ? context[ 0 ] : context;  
  
 // 当运行scripts为true时的向上兼容方案  
 // 如parseHTML不存在时会抛出错误  
 // $.merge() 函数用于合并两个数组内容到第一个数组。  
 // jQuery.parseHTML = function( data, context, keepScripts )  
 jQuery.merge( this, jQuery.parseHTML(  
 match[ 1 ],  
 context && context.nodeType ? context.ownerDocument || context : document,  
 true  
 ) );  
  
 // HANDLE: $(html, props)  
 // 根据html创建临时dom，如$('<div></div>',{id:'rextao'})  
 if ( rsingleTag.test( match[ 1 ] ) && jQuery.isPlainObject( context ) ) {  
 // 此时for..in循环，match表示的是context的key，对于$('<div></div>',{id:'rextao'})则是id  
 for ( match in context ) {  
  
 // 如context存在函数，则调用，如$('<div></div>',{id:'rextao',"click":function(){}})  
 if ( jQuery.isFunction( this[ match ] ) ) {  
 this[ match ]( context[ match ] );  
  
 // 不是函数则设置为属性  
 } else {  
 this.attr( match, context[ match ] );  
 }  
 }  
 }  
  
 return this;  
  
 // HANDLE: $(#id)  
 } else {//rquickExpr.exec("#id"),match[1]=undefined，match[2]=id  
 elem = document.getElementById( match[ 2 ] );  
  
 if ( elem ) {  
  
 // 将元素直接插入jQuery对象  
 this[ 0 ] = elem;  
 this.length = 1;  
 }  
 return this;  
 }  
  
 // HANDLE: $(expr, $(...)) ？？？？？？？？？？？？？？  
 // !context处理是的，如$("div"),此时上述match不存在，context也存在,在root域中寻找div  
 // context.jquery处理的是，context为jquery对象情况，  
 // 如$("div",$()),此时上述match不存在,context存在且是jQuery对象，  
 // 可以使用jQuery的find方法在context中查找div  
 } else if ( !context || context.jquery ) {  
 return ( context || root ).find( selector );  
  
 // HANDLE: $(expr, context)  
 // 相当于$(context).find(expr)  
 } else {// context不是jQuery对象，利用constructor将context构建为jQuery对象  
 return this.constructor( context ).find( selector );  
 }  
  
 // HANDLE: $(DOMElement)  
 } else if ( selector.nodeType ) {//当传入DOM 对象时，表明当前 jQuery 数组对象拥有一个元素  
 this[ 0 ] = selector;  
 this.length = 1;  
 return this;  
  
 // HANDLE: $(function)  
 // Shortcut for document ready,$传入一个函数时，会当dom加载完毕后立即运行此函数  
 } else if ( jQuery.isFunction( selector ) ) {  
 return root.ready !== undefined ?  
 // 如selector是function则在root下运行selector  
 // jQuery.fn.ready = function( fn ) {  
 root.ready( selector ) :  
  
 // 如root.ready函数不存在，则立即执行，参数为jquery对象  
 selector( jQuery );  
 }  
  
 return jQuery.makeArray( selector, this );//将selector与this合并  
};

# jQuery原型添加方法与属性

## pushStack----（L143）

为jQuery链式调用提供可能，所有可以链式调用的函数，都调用了pushStack函数，pushStack函数主要作用是将旧对象添加到新jquery对象的prevObject属性上

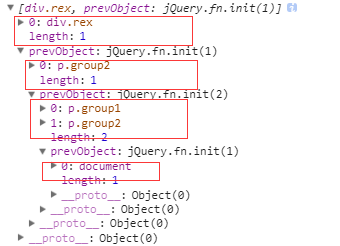
Html为

<div class="rex">  
　　<a>hello rextao</a>  
 <p class="group1">group1</p>  
　　<p class="group2">group2</p>  
　　<span class="group"></span>  
</div>  
<div class="aaa"></div>

如使用$('p').eq(1).parent()，因为eq与parent内部都调用了pushStack函数，根据html代码，可得到，$(‘p’)选择的是2个p元素，eq(1)则选择到p.group2，parent()则选择是div

.rex，可以看到，每次调用，都会把之前查询到的存储在prevObject中，比如开始为document，调用$(‘p’)，查询到p.group1与p.group2，则创建一个新的jQuery对象（例如a）,则a.preObject= document，然后调用.eq(1)，查询到p.group2，再创建新的jQuery对象（例如b），则b.preObject = a;一次类推

故可以假象为一个stack，每次查询到结果就push进去，因此stack=[document,[p.group1,p.group2],p.group2],最后查询到的div.rex并没有push到stack中，



源码分析

// 取一组元素并推入栈中，返回新匹配的元素集  
pushStack: function( elems ) {  
 // this.constructor = jquery，所以this.constructor()返回一个jQuery对象  
 var ret = jQuery.merge( this.constructor(), elems );  
  
 // 为ret添加prevObject属性，指向old对象  
 ret.prevObject = this;  
  
 // 返回新的ret元素集合  
 return ret;  
},

## end---(L184)

根据pushStack的分析，故每次调用end函数相当于，返回上述stack的最外层元素，即返回p.group2，每次调用弹出一个元素

源码

end: function() {

// this.constructor(),如当前this无prevObject属性，即非pushStack构建的，则返回当前的jQuery对象  
 return this.prevObject || this.constructor();  
},

## jQuery.fn = jQuery.prototype---(L114)

// 给JQ原型对象，添加一些方法和属性  
jQuery.fn = jQuery.prototype = {  
  
 // 当前的jQuery版本  
 jquery: version,  
  
 constructor: jQuery,  
  
 // jQuery对象的默认长度  
 length: 0,  
  
 toArray: function() {  
 return slice.call( this );  
 },  
  
 // 从匹配元素集中获取第N个元素或获取整个匹配元素集  
 // $('div').get(0),this则是html匹配到的dom元素  
 get: function( num ) {  
  
 // Return all the elements in a clean array  
 if ( num == null ) {  
 return slice.call( this );  
 }  
  
 // 如num<0,返回整个this，否则仅返回num这个元素  
 return num < 0 ? this[ num + this.length ] : this[ num ];  
 },  
  
 // 取一组元素并推入栈中，返回新匹配的元素集  
 // 类似于栈，将对象一层层包裹在prevObject属性上  
 pushStack: function( elems ) {  
 // this.constructor = jquery，所以this.constructor()返回一个jQuery对象  
 var ret = jQuery.merge( this.constructor(), elems );  
  
 // 为ret添加prevObject属性，指向old对象  
 ret.prevObject = this;  
  
 // 返回新的ret元素集合  
 return ret;  
 },  
  
 // 为每个匹配集合执行回调函数  
 each: function( callback ) {  
 return jQuery.each( this, callback );  
 },  
 // 对this的每个结果应用function函数，  
 map: function( callback ) {  
 return this.pushStack( jQuery.map( this, function( elem, i ) {  
 return callback.call( elem, i, elem );  
 } ) );  
 },  
 // 选取一个匹配的子集  
 slice: function() {  
 return this.pushStack( slice.apply( this, arguments ) );  
 },  
  
 first: function() {  
 return this.eq( 0 );  
 },  
  
 last: function() {  
 return this.eq( -1 );  
 },  
 // 对于$('div').eq(0),假如$('div')匹配到[div.rex,div.aaa]，this.length=2  
 // 因此this[0] = div.rex,传入pushStack处理  
 eq: function( i ) {  
 var len = this.length,  
 // ( i < 0 ? len : 0 ),处理eq(-2)情况，从数组最后往前计算  
 j = +i + ( i < 0 ? len : 0 );  
 // 主要是push哪个元素到stack（pushStac类似于一个栈）中  
 return this.pushStack( j >= 0 && j < len ? [ this[ j ] ] : [] );  
 },  
 // this.constructor(),如当前this无prevObject属性，即非pushStack构建的，则返回当前的jQuery对象  
 end: function() {  
 return this.prevObject || this.constructor();  
 },  
  
 // 内部使用  
 // 表现类似于数组方法，而不像jQuery方法  
 push: push,  
 sort: arr.sort,  
 splice: arr.splice  
};

# 核心函数

## extend

### 概述

jQuery大部分函数都是通过这个函数扩展而来的

### isPlainObject（判断纯粹对象）

* + - 1. 何为纯粹对象
         1. 由Object构造出来的对象
         2. new Object()所构造出来的对象，括号中没有任何内容，因为new Object(3);会构造一个Number类型的对象
         3. {}这种形式的对象，本质就是new Object();
         4. **原型链上直接继承Object.prototype的对象**

### toString、valueOf方法详解

* + - 1. 对于基本类型数据
         1. toString将数据转换为字符串
         2. valueOf返回原数据
      2. 深入toString的实现机制
         1. 根据官方说明，toString方法实际是返回this的类型，并以特殊字符串返回
      3. jQuery如何用toString实现数据类型判断的
         1. 开始声明class2type为空对象，用each方法将class2type转换为[object Boolean] = boolean形式，与toString返回值是一致的
         2. Type相当于用class2type[ toString.call( obj ) ]封装返回boolean
         3. 比使用typeof更细致

var class2type = {};  
var toString = class2type.toString;  
jQuery.each( "Boolean Number String Function Array Date RegExp Object Error Symbol"  
 .split( " " ),  
 function( i, name ) {  
 class2type[ "[object " + name + "]" ] = name.toLowerCase();  
 } );  
function type(obj) {  
 if ( obj == null ) {  
 return obj + "";  
 }  
 return typeof obj === "object" || typeof obj === "function" ?  
 class2type[ toString.call( obj ) ] || "object" :  
 typeof obj;  
}

* + - 1. class2type[ toString.call( obj ) ] || "object"
         1. 对于obj=navigator，toString.call( obj )会返回[object Navicator]，但class2type并没有Navictor这个值，故前半部分会返回undefined，结果返回object

### 源码分析

* + - 1. 在传入一个参数时，会将target=this，因此，对于jQuery.extend与jQuery.fn.extend的this指向不同，扩展对象也不同

//合并两个或更多对象的属性到第一个对象中，jQuery后续的大部分功能都通过该函数扩展  
jQuery.extend = jQuery.fn.extend = function() {  
 var options, name, src, copy, copyIsArray, clone,  
 target = arguments[ 0 ] || {},// 常见用法 jQuery.extend( obj1, obj2 )，此时，target为arguments[0]  
 i = 1,  
 length = arguments.length,  
 deep = false;  
  
 // 处理深复制情况  
 if ( typeof target === "boolean" ) {// 如果第一个参数为true，即 jQuery.extend( true, obj1, obj2 ); 的情况  
 deep = target; // 此时target是true  
  
 // Skip the boolean and the target  
 target = arguments[ i ] || {}; //target为obj1  
 i++; //i=2  
 }  
  
 // 如第一参数不是obj或函数，如jQuery.extend("haa",obj1,obj2)  
 if ( typeof target !== "object" && !jQuery.isFunction( target ) ) {  
 target = {};  
 }  
  
 // 只传入一个参数，处理jQuery.extend(obj)，或 jQuery.fn.extend( obj )  
 if ( i === length ) {  
 target = this;//对于extend和fn.extend的this指向不同，  
 i--; //i=0;  
 }  
  
 for ( ; i < length; i++ ) {  
  
 // 处理非空参数  
 if ( ( options = arguments[ i ] ) != null ) {  
  
 for ( name in options ) {  
 src = target[ name ]; // arguments[ 0 ]  
 copy = options[ name ];  
  
 // 避免循环引用\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 // 属性拷贝要考虑循环引用的问题，否则会一直复制  
 if ( target === copy ) {  
 continue; //continue 用于跳过当前循环  
 }  
  
 // 如是深拷贝，  
 if ( deep && copy && ( jQuery.isPlainObject( copy ) ||  
 ( copyIsArray = Array.isArray( copy ) ) ) ) {  
  
 if ( copyIsArray ) {  
 copyIsArray = false;  
 clone = src && Array.isArray( src ) ? src : [];  
  
 } else {  
 clone = src && jQuery.isPlainObject( src ) ? src : {};  
 }  
  
 // Never move original objects, clone them  
 target[ name ] = jQuery.extend( deep, clone, copy );//递归  
  
 // 浅拷贝，copy不为undefined的值  
 } else if ( copy !== undefined ) {  
 target[ name ] = copy;  
 }  
 }  
 }  
 }  
  
 // 返回修改后的对象  
 return target;  
};

## 工具函数

### isArrayLike()

* + - 1. 拥有数组的一部分行为
      2. In操作符两种用法
         1. For…in..判断数组或对象迭代
         2. 判断对象是否为数组/对象的元素/属性

——至于（length-1）in obj 干啥的是不知道？？？？？？？？？？？？？？？？

* + - 1. 具有如下行为成为类数组：
         1. 具有length属性
         2. 按索引方式存储数据
         3. 不具有数组的push,pop等方法

function isArrayLike( obj ) {  
 // && 第一个操作数能强转为真，则返回第二个操作数，无论真假  
 var length = !!obj && "length" in obj && obj.length,  
 type = jQuery.type( obj );  
 // window对象具有length属性，function函数length属性指的参数列表  
 if ( type === "function" || jQuery.isWindow( obj ) ) {  
 return false;  
 }  
 // typeof length === "number"表示获得obj.length  
 return type === "array" || length === 0 ||  
 typeof length === "number" && length > 0 && ( length - 1 ) in obj;  
}

### $.globalEval()

* + - 1. 把一段脚本加载到全局context（window）中。
      2. 因为使用jQuery的环境为jQuery，如想将上下文环境设置为window，需要用到个函数
      3. 实际就是在document创建script标签，运行代码后，将script标签删除，如不删除标签，每次调用都添加，会造成页面过多标签

globalEval: function( code ) {  
 DOMEval( code );  
}

function DOMEval( code, doc ) {  
 doc = doc || document;  
 var script = doc.createElement( "script" );  
 script.text = code; doc.head.appendChild( script ).parentNode.removeChild( script );  
}

### $.camelCase()

//匹配破坏驼峰标记法的字符串  
rmsPrefix = /^-ms-/,//ie前缀  
rdashAlpha = /-([a-z])/g,  
  
fcamelCase = function( all, letter ) {  
 return letter.toUpperCase();  
};

// 转换为驼峰标记法  
// Support: IE <=9 - 11, Edge 12 - 13  
// rmsPrefix = /^-ms-/,  
// rdashAlpha = /-([a-z])/g,  
// 相当于把rex-tao转换为rexTao  
camelCase: function( string ) {  
 return string.replace( rmsPrefix, "ms-" ).replace( rdashAlpha, fcamelCase );  
},

### $.isNumeric()

* + - 1. Obj必须为number或string类型
      2. 注意：isNaN有个bug，isNaN(“aaa”)会返回true，以及parseFloat(“123a”)可以返回123，故isNaN(parseFloat(obj))是不能检测正确的

isNumeric: function( obj ) {  
 var type = jQuery.type( obj );  
 return ( type === "number" || type === "string" ) &&  
 !isNaN( obj - parseFloat( obj ) );  
}

### $.isEmptyObject()

// 是否空对象  
isEmptyObject: function( obj ) {  
 var name;  
  
 for ( name in obj ) {  
 return false;  
 }  
 return true;  
},

### $.type()

* + - 1. Class2type定义

// 创建类型对应关系  
// tostring.call('a'),会返回[object String]，通过class2type则可以返回string,方便类型判断  
jQuery.each( "Boolean Number String Function Array Date RegExp Object Error Symbol".split( " " ),  
function( i, name ) {  
 class2type[ "[object " + name + "]" ] = name.toLowerCase();  
} );

* + - 1. 源码

// 判断类型  
type: function( obj ) {  
 if ( obj == null ) {  
 return obj + "";  
 }  
 // Line518行对此对象进行了充实，并不是空对象  
 // —— 对于obj=navigator，toString.call( obj )会返回[object Navicator]，但class2type并没有Navictor这个值，故前半部分会返回undefined，结果返回object  
 return typeof obj === "object" || typeof obj === "function" ?  
 class2type[ toString.call( obj ) ] || "object" :  
 typeof obj;  
}

### $.each()

* + - 1. callback.call( obj[ i ], i, obj[ i ] ) 是callback返回值，如果返回值为flase，则不将前面的obj运用到callback上
      2. call第一个参数为绑定对象，之后为参数列表

each: function( obj, callback ) {  
 var length, i = 0;  
 if ( isArrayLike( obj ) ) {  
 length = obj.length;  
 for ( ; i < length; i++ ) {   
 if ( callback.call( obj[ i ], i, obj[ i ] ) === false )

{break;}  
 }  
 } else {  
 for ( i in obj ) {  
 if ( callback.call( obj[ i ], i, obj[ i ] ) === false )

{break;}  
 }  
 }  
 return obj;  
},

### $.merge()

* + - 1. 将数组second合并到数组first中
      2. 普通对象并无length属性，如手动添加，但非类数组，故[]取不到元素，只能返回undefined

merge: function( first, second ) {  
 var len = +second.length,  
 j = 0,  
 i = first.length;  
 for ( ; j < len; j++ ) {  
 first[ i++ ] = second[ j ];  
 }  
 // 指定数组长度，程序运行时会分配空间，降低运行速度  
 // 主要是考虑到后期代码内存的优化问题。  
 first.length = i;  
 return first;  
},

### $.grep()

* + - 1. 过滤数组，返回新数组,invert为true时，callback返回false，才添加到数组中

grep: function( elems, callback, invert ) {  
 var callbackInverse,  
 matches = [],  
 i = 0,  
 length = elems.length,  
 callbackExpect = !invert;  
  
 for ( ; i < length; i++ ) {  
 callbackInverse = !callback( elems[ i ], i );  
 if ( callbackInverse !== callbackExpect ) {  
 matches.push( elems[ i ] );  
 }  
 }  
  
 return matches;  
}

### $.map()

* + - 1. 让elems分别作为参数传入callback中，返回数组
      2. 主要注意最后利用apply返回展开数组

map: function( elems, callback, arg ) {  
 var length, value,  
 i = 0,  
 ret = [];  
 // 转换类数组  
 if ( isArrayLike( elems ) ) {  
 length = elems.length;  
 for ( ; i < length; i++ ) {  
 value = callback( elems[ i ], i, arg );  
 if ( value != null ) {  
 ret.push( value );  
 }  
 }  
 // 如是对象  
 } else {  
 for ( i in elems ) {  
 value = callback( elems[ i ], i, arg );  
  
 if ( value != null ) {  
 ret.push( value );  
 }  
 }  
 }  
 // 展开数组ret，利用apply特性  
 // 可能ret=[1,[2,3]],apply会将数组转换为一个个参数  
 return concat.apply( [], ret );  
}

### $.trim()

* + - 1. \uFEFF是utf8的字节序标记
         1. Unicode规范中有一个BOM的概念，BOM——Byte Order Mark，就是字节序标记
         2. \uFEFF是规范在传递数据的一个数据头
      2. \xA0″是全角空格
      3. Rtrim这个匹配，利用|运算符，分别匹配开头的空格和结尾空格

// rtrim = /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g  
trim: function( text ) {  
 return text == null ?  
 "" :  
 ( text + "" ).replace( rtrim, "" );  
}

### $.proxy()

* + - 1. 调用方式
         1. $.proxy(function,context)
         2. $.proxy(context,name),context为一个对象，name为对象属性并且是一个函数
         3. jQuery.proxy( function, context [, additionalArguments ] )，将参数传给functrion
      2. 函数调用

var objPerson = {  
 name: "obj",  
 age: 32,  
 test: function(a,b,c) {  
 $("p").after("Name: " + this.name + "<br> Age: " + this.age+ "<br> addArguments: "+a+b+c);  
 }  
};  
$("#test").on("click", $.proxy(objPerson.test, objPerson));  
$("#test").on("click", $.proxy(objPerson, "test"));  
$("#test").on("click", $.proxy(objPerson, "test","hello","rextao","!!"));

* + - 1. jQuery源码

proxy: function( fn, context ) {  
 var tmp, args, proxy;  
 // 处理第2种调用方式

if ( typeof context === "string" ) {  
 tmp = fn[ context ];  
 context = fn;  
 fn = tmp;  
 }  
 // 快速测试fn是否是可调用的（即函数），在文档说明中，会抛出一个TypeError，

// 但是这里仅返回undefined

if ( !jQuery.isFunction( fn ) ) {  
 return undefined;  
 }  
 // 处理第3种调用方式

args = slice.call( arguments, 2 ); // 从参数列表中去掉fn,context

proxy = function() {  
 return fn.apply( context || this, args.concat( slice.call( arguments ) ) );  
 };  
 // 统一guid，使得proxy能够被移除

proxy.guid = fn.guid = fn.guid || jQuery.guid++;  
 return proxy;  
},

## 将jQuery变为可迭代对象---( L512)

* + - 1. Jquery是类数组对象，故可以将jquery对象的Symbol.iterator属性布置为arr[Symbol.iteratro]，arr为[]，这样jquery对象为可迭代对象，可以将jquery用于for..of循环
      2. 自定义对象，配置Symbol.iterator属性，有要求，如@@iterator要返回一个迭代器对象，不符合要求的会抛出异常

// Symbol.iterator 为对象定义了默认的迭代器  
if ( typeof Symbol === "function" ) {  
 jQuery.fn[ Symbol.iterator ] = arr[ Symbol.iterator ];  
}

* + - 1. 自定义迭代器

var myIterable = {}  
myIterable[Symbol.iterator] = function\* () {  
 yield 1;  
 yield 2;  
 yield 3;  
};  
[...myIterable] // [1, 2, 3]