# 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

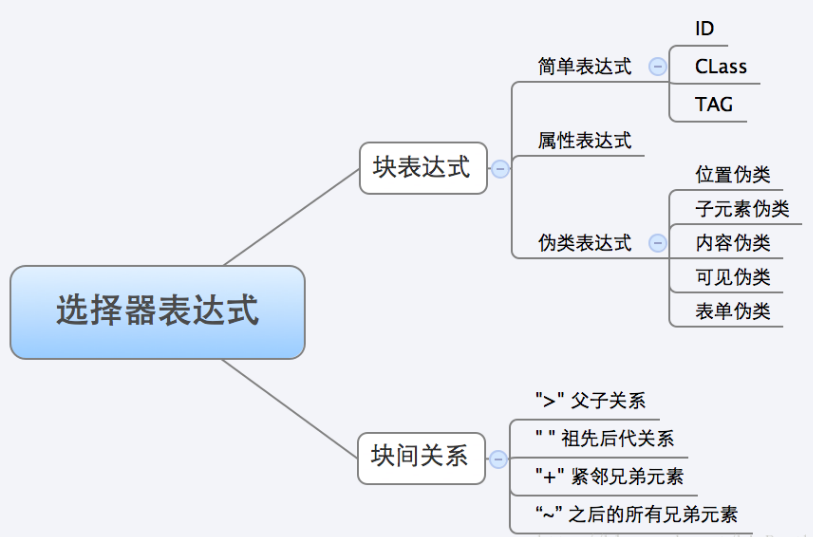
## Sizzle的设计思路

### 概述

* + - 1. 对于一个复杂的选择器表达式（前提是浏览器不支持querySelectorAll），如何对其进行处理？
      2. jQuery的特点快捷选取、操作DOM元素，如#id .class > p:first-child, .class2 input[type=’text’]，但js本身名没有这么复杂的选择器，只有getElementById，getElementByTagName等，作者写了一个Sizzle引擎，用于css选取
      3. Sizzle这样的选择器引擎是为了解决某些低级浏览器没有document.getElementsByClassName，document.querySelector，document.querySelectorAll接口

### 分割解析

* + - 1. 对于$(“#id .class > p:first-child, .class2 input[type=’text’]”)，可以处理的办法有什么呢？
         1. split(',')来处理，处理出来的仅是单独块，不能获得更多信息
         2. 正则分块，获取结果的同时可以提取信息，缺点是可读性差以及效率问题
      2. jQuery处理办法
         1. 对于选择器表达式#id .class > p.red + a, .class2 p:first-child input[type=’text’]拆分为块表达式、块间表达式
         2. 对于有逗号（，）的，是先分离出#id .class > p.red + a，处理完毕后再分离.class2 p:first-child input[type=’text’]处理
         3. 故对于#id .class > p.red + a进行处理，得到块表达式A：#id .class；B: p.red；C：a，块间关系为A>B，B+C



### 块表达式内部查找

* + - 1. 对于上述块表达式ABC，只能使用DOM中的API查找
      2. 代码的效率是一个永恒的主题，因此一定是ID > Class > Name> Tag
      3. 对于块表达式A，会先查询#id得到集合a，然后再集合a中筛选出.class的集合，因为分别查找#id和.class是完全没必要的

### 块间关系处理

* + - 1. 处理了块表达式内部查找，需要处理的关键是块间关系，处理方式可以从左到右，也可从右到左
      2. 情况一、body div p:first===》(body div p):first
         1. 自左向右的查询。先找到body，获得集合A，然后在集合A中查找div获得集合B，在集合B中查找p获得集合C，最后取集合C的第一个元素，得最终结果XX
      3. 情况二、body div p:first-child
         1. 自右向左的查询，先找到p:first-child获得集合A1，然后判断祖先元素里面是否有div获得集合B1，然后判断祖先元素里面是否有body获得集合C1，得最终结果C1
      4. 情况三、div#a p
         1. 先找到div#a，获得单个元素A1，然后再A1的环境中查找p，得最终结果B1

## 正则表达式(613)

### whitespace(匹配空白符)

* + - 1. 首先要特别注意，写在前面的是变量，故有两个\\，最好输出正则后再看具体含义
      2. <https://regexper.com/>，正则图示软件，感觉很不错，方便分析复杂正则表达式
      3. <https://www.debuggex.com/>：可以调试正则表达式，判断给定的string是否满足正则

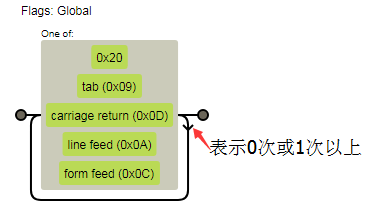
whitespace = "[\\x20\\t\\r\\n\\f]"

rwhitespace = new RegExp( whitespace + "+", "g" )

* + - * 1. 结果：/[\x20\t\r\n\f]+/g
      1. 可以通过查询ASCII对照表，看十六进制\x对应的是什么
         1. \x20：空白符
         2. \x0D：\r归为符

—— \x0A：\n换行符

* + - * 1. \0x0C:\f换页符



### Identifier(匹配标识符)

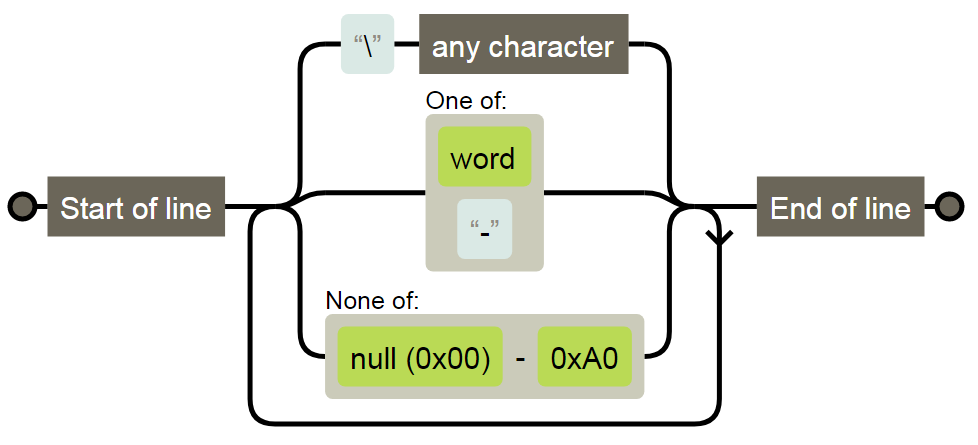
* + - 1. w3c文档规定，css的标识符只能包含[a-zA-Z0-9]和ISO 10646标准中的U+00A0和它之后的字符，以及中划线、下划线。不能以数字，两个中划线或者中划线后跟一个数字作为标识符的开始。标识符也可以包含转义字符和任意ISO 10646作为数字编码。例如标识符“B&W?”可以写成“B\26 W\3F”。
      2. \0-\xa0：为了满足W3C中字符要是“U+00A0”以及其后的字符的要求，排除ISO 10646编码中0-xa0范围内的字符都排除
      3. \w：匹配单词，等于[a-zA-Z0-9]
      4. \0：表示的是C0控制符中的NULL

注意：下面结果中，红字^和-中间有一个空格，在浏览器中显示为一个空格了，实际是\0

identifier = "(?:\\\\.|[\\w-]|[^\0-\\xa0])+"

ridentifier = new RegExp( "^" + identifier + "$" )

* + - * 1. Chrome控制台结果：/^(?:\\.|[\w-]|[^ -\xa0])+$/
        2. 实际结果：/^(?:\\.|[\w-]|[^\0-\xa0])+$/

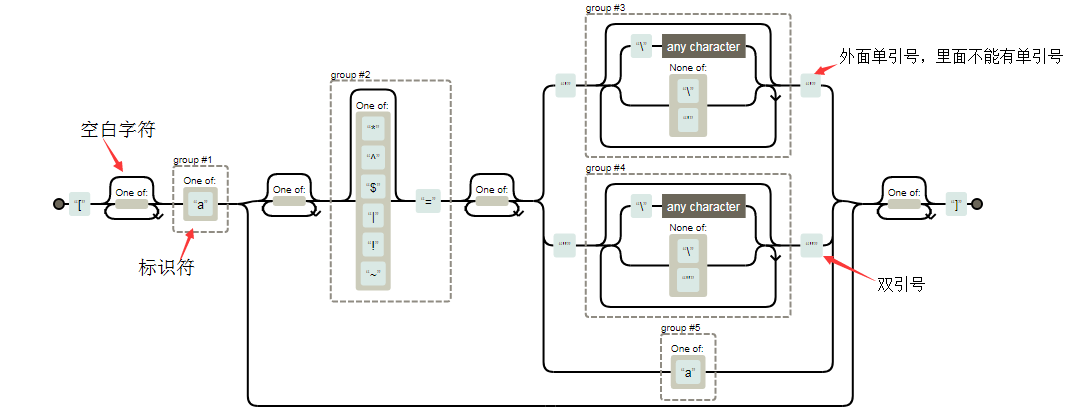


### Attributes(匹配属性)

* + - 1. 通过图示，可将复杂的正则很快区分出来

attributes = "\\[" + whitespace + "\*(" + identifier + ")(?:" + whitespace +  
 // Operator (capture 2)  
 "\*([\*^$|!~]?=)" + whitespace +  
 // "Attribute values must be CSS identifiers [capture 5] or strings [capture 3 or capture 4]"  
 "\*(?:'((?:\\\\.|[^\\\\'])\*)'|\"((?:\\\\.|[^\\\\\"])\*)\"|(" + identifier + "))|)" + whitespace +  
 "\*\\]"

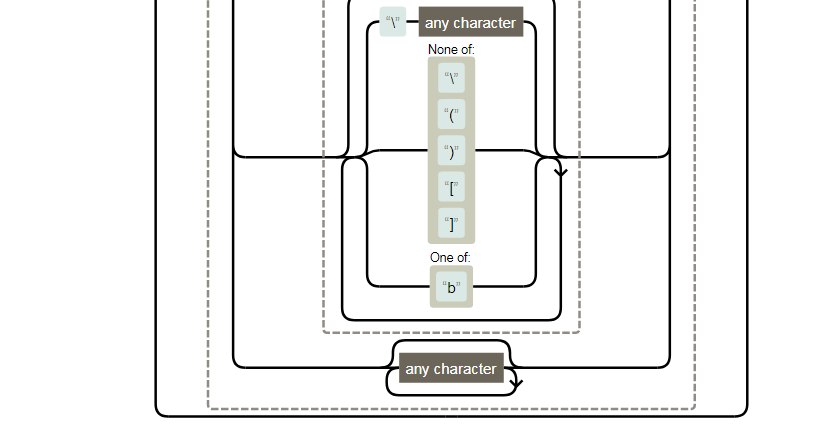
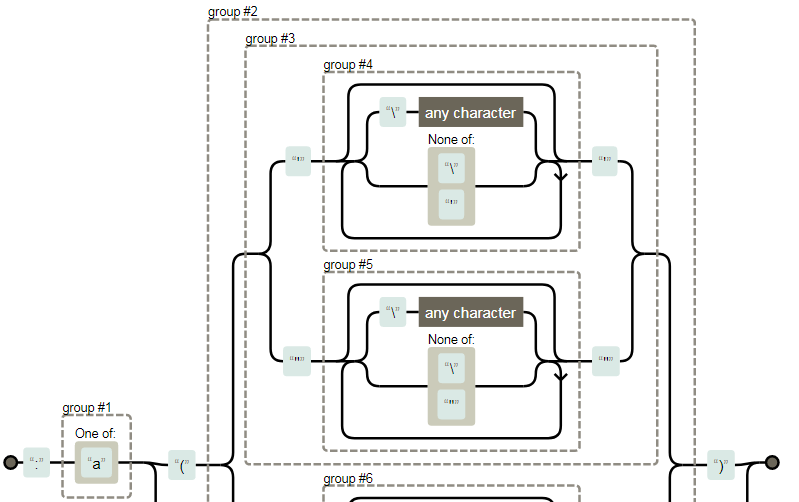
* + - 1. 为了结构更清晰，将whitespace设为[]，identifier设置为[a]
         1. /\[[]\*([a])(?:[]\*([\*^$|!~]?=)[]\*(?:'((?:\\.|[^\\'])\*)'|"((?:\\.|[^\\"])\*)"|([a]))|)[]\*\]/



### Pseudos(匹配伪类)

* + - 1. 将identifer设置为[a]，将attributes设置为[b]
         1. /:([a])(?:\((('((?:\\.|[^\\'])\*)'|"((?:\\.|[^\\"])\*)")|((?:\\.|[^\\()[\]]|[b])\*)|.\*)\)|)/

pseudos = ":(" + identifier + ")(?:\\((" +  
 // To reduce the number of selectors needing tokenize in the preFilter, prefer arguments:  
 // 1. quoted (capture 3; capture 4 or capture 5)  
 "('((?:\\\\.|[^\\\\'])\*)'|\"((?:\\\\.|[^\\\\\"])\*)\")|" +  
 // 2. simple (capture 6)  
 "((?:\\\\.|[^\\\\()[\\]]|" + attributes + ")\*)|" +  
 // 3. anything else (capture 2)  
 ".\*" +  
 ")\\)|)"

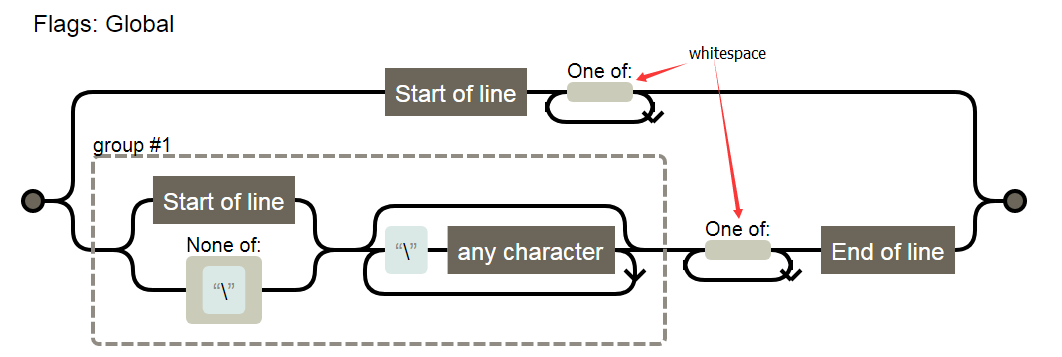


### rtrim

* + - 1. 将whitespace设置为[]
      2. 匹配字符串两头的空白

rtrim = new RegExp( "^" + whitespace + "+|((?:^|[^\\\\])(?:\\\\.)\*)" + whitespace + "+$", "g" )

* + - * 1. 结果：/^[]+|((?:^|[^\\])(?:\\.)\*)[]+$/g



### rcomma

* + - 1. rcomma = /^[\x20\t\r\n\f]\*,[\x20\t\r\n\f]\*/,匹配（ ， ）形式

// rcomma = new RegExp( "^" + whitespace + "\*," + whitespace + "\*" )

### rcombinators

* + - 1. 匹配是否为关系符，>+~空格

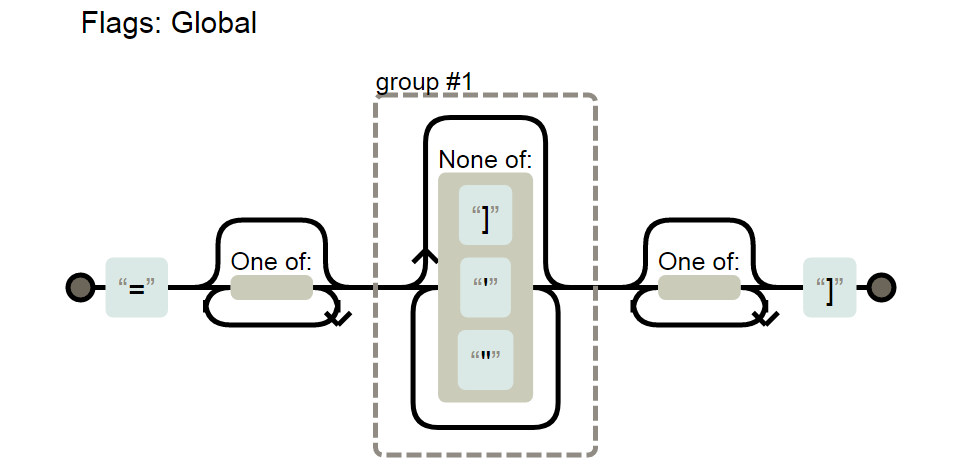
rcombinators = new RegExp( "^" + whitespace + "\*([>+~]|" + whitespace + ")" + whitespace + "\*" )

### rattributeQuotes(匹配属性)

* + - 1. \*?表示懒惰模式，匹配能匹配到最少词，.\*匹配的最长词
      2. 对于字符串aabab，a.\*?b匹配的是aab，a.\*b匹配的是aabab

rattributeQuotes = new RegExp( "=" + whitespace + "\*([^\\]'\"]\*?)" + whitespace + "\*\\]", "g" )

* + - * 1. 结果：/=[]\*([^\]'"]\*?)[]\*\]/g

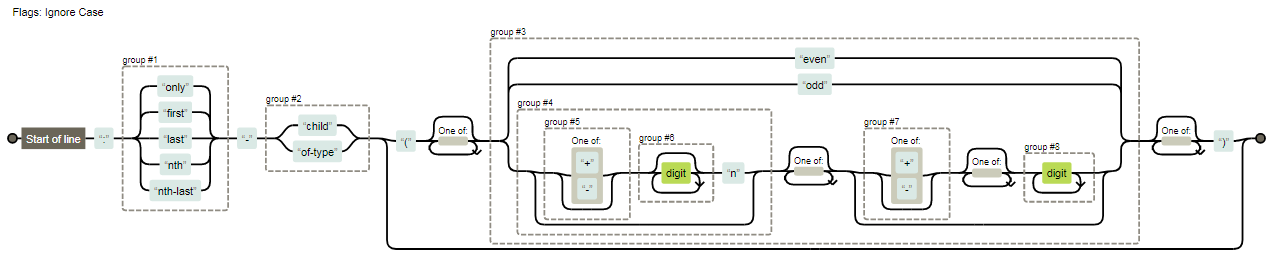


### matchExpr. CHILD

"CHILD": new RegExp( "^:(only|first|last|nth|nth-last)-(child|of-type)(?:\\(" + whitespace +  
 "\*(even|odd|(([+-]|)(\\d\*)n|)" + whitespace + "\*(?:([+-]|)" + whitespace +  
 "\*(\\d+)|))" + whitespace + "\*\\)|)", "i" )

* + - * 1. 结果：（空白使用[]替换）

/^:(only|first|last|nth|nth-last)-(child|of-type)(?:\([]\*(even|odd|(([+-]|)(\d\*)n|)[]\*(?:([+-]|)[]\*(\d+)|))[]\*\)|)/i



* + - 1. 可以匹配如下表达式
         1. :only-child
         2. :only-child( )
         3. :only-child( +123n )
         4. :only-child(+123n -1234)

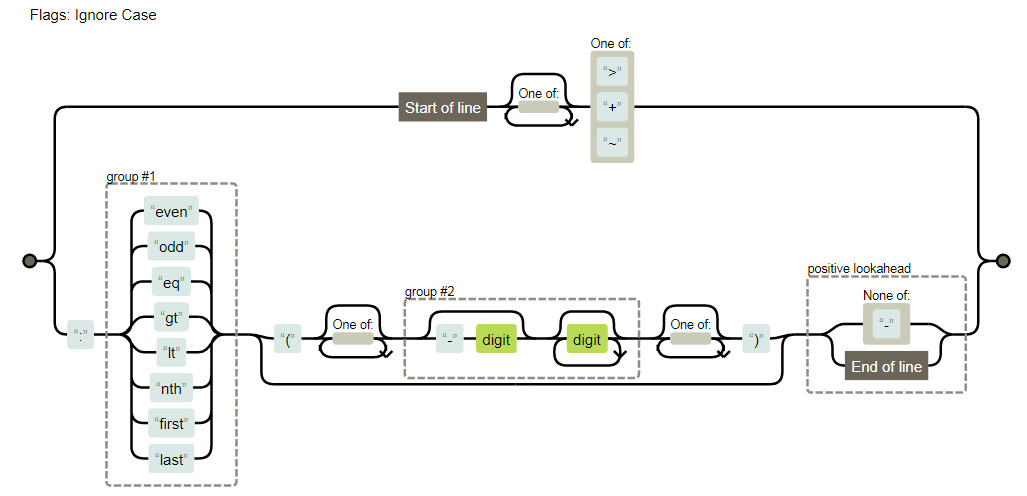
### matchExpr.needsContext

* + - 1. 用于实现.is()
      2. 用于select = Sizzle.select中，判断第一个token是[>+~]关系选择器，或整个选择器中有sizzle自定义伪类

"needsContext": new RegExp( "^" + whitespace + "\*[>+~]|:(even|odd|eq|gt|lt|nth|first|last)(?:\\(" +  
 whitespace + "\*((?:-\\d)?\\d\*)" + whitespace + "\*\\)|)(?=[^-]|$)", "i" )

* + - * 1. 结果：（空白使用[]替换）

/^[]\*[>+~]|:(even|odd|eq|gt|lt|nth|first|last)(?:\([]\*((?:-\d)?\d\*)[]\*\)|)(?=[^-]|$)/i



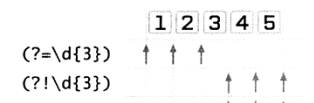
### 补：?=与?!

* + - 1. 环视语法（详见《正则指引p91》）
      2. ？= 肯定顺序环视，判断方向右，结构内表达式匹配成功，返回true

！= 否定顺序环视，判断方向右，结构内表达式匹配成功，返回false

var a1 = /(?=\d{3})/g;  
var a2 = /(?!\d{3})/g;  
console.log('123456'.replace(a1,',')); //——,1,2,3,456  
console.log('123456'.replace(a2,',')); //—— 1234,5,6,

* + - 1. 匹配规则图示，如？=开始位置在1前面，然后向后环视，看后面是否有3个数字，确定有则将此处添加一个逗号，1前面；然后向右移位到2前面，然后再环视。。当到4前面时，后面的数字不满足\d{3}返回false



### rnative

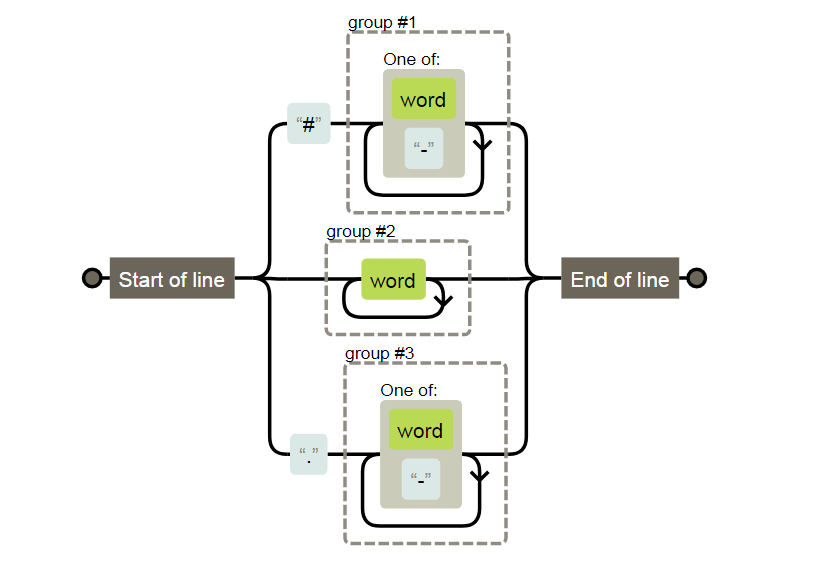
/^[^{]+\{\s\*\[native \w/

* + - 1. 用以匹配function getElementByClassName(){[native code}代码，判断一个函数是否存在

### rquickExpr

* + - 1. 快速解析ID，tag，class

/^(?:#([\w-]+)|(\w+)|\.([\w-]+))$/,



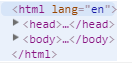
## setDocument(1108)

### documentElement与ownerDocument区别

* + - 1. ownerDocument：返回当前节点的顶层的document对象，环境不是window，可能返回不是window.document
         1. 返回的是一个对象
         2. console.log(a.ownerDocument);返回的结果



* + - 1. documentElement：返回文档对象（document）的根元素的只读属性
         1. 如html文档中的<html>元素
         2. console.log(document.documentElement);



### isXML = Sizzle.isXML(1096)

* + - 1. 检测是否为xml节点
      2. 根据：<https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeName>，介绍
         1. Xhtml或其他xml格式，document.getElementById("d1").nodeName为div（全部小写），在html中为DIV（全部大写）

/\*\*  
 \* 检测xml节点  
 \* @param {Element|Object} elem： 一个elment或document  
 \* @returns {Boolean} True ：如为true则不是html节点  
 \*/  
isXML = Sizzle.isXML = function( elem ) {  
 // documentElement返回文档  
 var documentElement = elem && (elem.ownerDocument || elem).documentElement;  
 return documentElement ? documentElement.nodeName !== "HTML" : false;  
};

### assert (920)

* + - 1. 测试一个element是否符合fn要求，符合返回true，否则返回null
      2. ？？？？？？？？？？？？？？为何创建fieldset元素，用来测试

function assert( fn ) {  
 var el = document.createElement("fieldset");  
 try {  
 return !!fn( el );  
 } catch (e) {  
 return false;  
 } finally {  
 // Remove from its parent by default  
 if ( el.parentNode ) {  
 el.parentNode.removeChild( el );  
 }  
 // 释放内存  
 el = null;  
 }  
}

* + - 1. 为何使用try-catch结构：
         1. 如源码L1313，解决某个bug，使用断言函数，此时如下代码IE8会报错

assert(function( el ) {

……

if ( !el.querySelectorAll(":checked").length ) {  
 rbuggyQSA.push(":checked");  
 }

}

### 区分properties和attributes

* + - 1. 元素特性attribute是指HTML元素标签的特性
         1. 下面的id、class、title、a都是特性，其中a叫做自定义特性

<div id="id1" class="class1" title="title1" a='a1'></div>

* + - 1. 对象属性property是指元素节点的属性
         1. 下面的id、title、b都是属性，其中b叫做自定义属性

<script>  
 var test = document.getElementById('text');  
 test.id = 'id2';  
 test.title = 'title2';  
 test.b = 'b2';  
</script>

* + - 1. 支持IE<8（1143）
         1. 官方介绍：验证getAttribute实际是返回attributes而不是properties

support.attributes = assert(function( el ) {  
 el.className = "i";  
 return !el.getAttribute("className");  
});

* + - * 1. 对于如下，代码，在普通浏览器中返回的是123，在IE8返回的是properties，即rextao
        2. 对于IE<8，如上代码，el.getAttribute("className")会返回i，而其他代码会返回null；
        3. 对于IE<8，如上代码，el.getAttribute('class')会返回null，其他的会返回i

<input type="text" value="123" id="input" title="input" />  
var a = document.getElementById("input");  
a.value = "rextao";  
console.log(a.getAttribute("value"));

### Element.matches()-1360

* + - 1. 如元素被指定的选择器字符串选择返回true，否则返回false
      2. 一些浏览器以非标准是方式，前缀+matchesSelector()实现了这个方法

<div id="id" class="c1"><span class="c2"></span></div>  
<script src="jquery-3.2.1.js"></script>  
<script>  
 var docElem =document.getElementById('id');  
 console.log(docElem.matches('div')); // true  
 console.log(docElem.matches('.c1')); // true  
 console.log(docElem.matches('.c2')); // false  
</script>

### Node.compareDocumentPosition(otherNode)-1383

* + - 1. 比较两个节点，返回描述他们位置关系的整数，整数值其实是用bits值表示的，故可通过&位运算来判断何种类型

名称 值 bits

没关系，不属于同一文档 1 000001

第1节点在第二节点后 2 000010

第1节点在第二节点前 4 000100

第1节点在第二节点内 8 001000

第1节点在第二节点外 16 010000

没关系，浏览器私有使用 32 100000

* + - 1. Contains的兼容性解决办法

contains = hasCompare || rnative.test( docElem.contains ) ?  
 function( a, b ) {  
 // 如a是DOM根节点，返回顶层document对象  
 var adown = a.nodeType === 9 ? a.documentElement : a,  
 bup = b && b.parentNode;  
 // 如a包含b，则b.parentNode为a或a的子  
 return a === bup || !!( bup && bup.nodeType === 1 && (  
 adown.contains ? // 如contais方法存在，使用contains判断  
 adown.contains( bup ) :  
 // a.compareDocumentPosition( bup ) ==16表示a为bup的外部节点，如前2个&&为真，返回16  
 a.compareDocumentPosition && a.compareDocumentPosition( bup ) & 16  
 ));  
 } :  
 // 不存在contains和compareDocumentPosition方法，使用parentNode一层层比较  
 function( a, b ) {  
 if ( b ) {  
 while ( (b = b.parentNode) ) {  
 if ( b === a ) {  
 return true;  
 }  
 }  
 }  
 return false;  
 };

### siblingCheck()——954

* + - 1. sourceIndex：IE的属性，表示对象在文档出现顺序的序列

/\*\*  
 \* 检查同辈元素document顺序  
 \* @param {Element} a  
 \* @param {Element} b  
 \* @returns {Number} 返回值小于0，a在b前，大于0，则a在b后  
 \*/  
function siblingCheck( a, b ) {  
 var cur = b && a,  
 //  
 diff = cur && a.nodeType === 1 && b.nodeType === 1 &&  
 a.sourceIndex - b.sourceIndex;

// 如存在sourceIndex，使用IE的sourceIndex  
 if ( diff ) {  
 return diff;  
 }

// 检查b是否在a后面  
 if ( cur ) {  
 while ( (cur = cur.nextSibling) ) {  
 if ( cur === b ) {  
 return -1; //a在b前面  
 }  
 }  
 }  
 // 此句目的？？？？？？？？？？？？？

// 为了让函数返回1或-1，如a为null，返回 -1？？？？？  
 return a ? 1 : -1;  
}

### 完整代码

setDocument = Sizzle.setDocument = function( node ) {  
 var hasCompare, subWindow,  
 // preferredDoc = window.document ,返回window的document引用，是一个对象  
 // ownerDocument,返回当前节点的顶层的document对象，环境不是window，可能返回不是window.document  
 doc = node ? node.ownerDocument || node : preferredDoc;  
  
 // nodeType ==9：一个 Document 节点  
 // documentElement：返回文档对象（document）的根元素的只读属性  
 if ( doc === document || doc.nodeType !== 9 || !doc.documentElement ) {  
 return document; //返回window.document  
 }  
  
 // 更新全局变量  
 document = doc;  
 docElem = document.documentElement;  
 documentIsHTML = !isXML( document );  
  
 // Support: IE 9-11, Edge  
 // 解决iframe文件卸载后，抛出“没有权限”的bug (jQuery #13936)  
 // document.defaultView:在浏览器中返回document相关的window对象  
 // ？？？？？？？？？？？为何能通过这种方式修复bug-。-不需要关注  
 if ( preferredDoc !== document &&  
 (subWindow = document.defaultView) && subWindow.top !== subWindow ) {  
  
 // Support: IE 11, Edge  
 if ( subWindow.addEventListener ) {  
 subWindow.addEventListener( "unload", unloadHandler, false );  
  
 // Support: IE 9 - 10 only  
 } else if ( subWindow.attachEvent ) {  
 subWindow.attachEvent( "onunload", unloadHandler );  
 }  
 }  
  
 /\* 是否支持属性  
 // 其实support为true，表示当前方法或属性没有bug，如果为false表示有bug，需要特殊处理  
 ---------------------------------------------------------------------- \*/  
  
 // Support: IE<8  
 // 在使用getAttribute('class')时,IE<8,需要写为getAttribute('className')，IE8修复  
 // <input type="text" value="123" id="input" title="input" />  
 // var a = document.getElementById("input");  
 // a.value = "rextao";  
 // console.log(a.getAttribute("value"));  
 //—— 对于IE<8，如上代码，el.getAttribute("className")会返回i，而其他代码会返回null；  
 support.attributes = assert(function( el ) {  
 el.className = "i";  
 return !el.getAttribute("className");  
 });  
  
 // 针对support情况，为Expr增加find与filter方法  
 /\* getElement(s)By\*  
 ---------------------------------------------------------------------- \*/  
  
 // Check if getElementsByTagName("\*") returns only elements  
 //IE<9会考虑上comment节点，返回1，其他返回0  
 support.getElementsByTagName = assert(function( el ) {  
 el.appendChild( document.createComment("") );  
 return !el.getElementsByTagName("\*").length;  
 });  
  
 // Support: IE<9  
 // IE<9,因为没有document.getElementsByClassName，故返回undefined，  
 // rnative = /^[^{]+\{\s\*\[native \w/; \w匹配单词字符  
 // 支持的返回 function getElementsByClassName() { [native code] }  
 support.getElementsByClassName = rnative.test( document.getElementsByClassName );  
  
 // Support: IE<10，  
 // 低版本IE，getElementById会返回表单元素是name值  
 // <input type="text" name="myElement" value="The field" />  
 // <div id="myElement">A div</div> 对于这个代码，会返回input元素  
 // expando = "sizzle" + 1 \* new Date(),  
 // 对于非IE，getElementsByName( expando ).length=0，因为设定是的id，不会查到  
 support.getById = assert(function( el ) {  
 docElem.appendChild( el ).id = expando;  
 return !document.getElementsByName || !document.getElementsByName( expando ).length;  
 });  
  
 // ID filter and find  
 // filter查看是否有某个id元素  
 // find 查找某个元素  
 if ( support.getById ) {  
 Expr.filter["ID"] = function( id ) {  
 //解决firefox<24错误的数字解读，有w3c规范说明=。=  
 var attrId = id.replace( runescape, funescape );  
 return function( elem ) {  
 return elem.getAttribute("id") === attrId;  
 };  
 };  
 //查询id  
 Expr.find["ID"] = function( id, context ) {  
 if ( typeof context.getElementById !== "undefined" && documentIsHTML ) {  
 var elem = context.getElementById( id );  
 return elem ? [ elem ] : [];  
 }  
 };  
 } else {  
 Expr.filter["ID"] = function( id ) {  
 var attrId = id.replace( runescape, funescape );  
 return function( elem ) {  
 //返回指定元素的指定属性， 返回值是 Attr 节点类型，此方法是已废弃方法  
 //MDN上说已经使用getAttribute代替，并且性能上， element.id >element.getAttribute('id') >element.getAttributeNode('id').nodeValue.  
 // 但是对于如下代码，对于IE<7,下面1行返回null，而2可以返回rextao  
 // var el = document.getElementById('myElement')  
 // el.className ='rextao'  
 // console.log(el.getAttribute('class'))------------1  
 // console.log(el.getAttributeNode('class').value)------2  
 var node = typeof elem.getAttributeNode !== "undefined" &&  
 elem.getAttributeNode("id");  
 return node && node.value === attrId;  
 };  
 };  
  
 // Support: IE 6 - 7 only, getElementById不可靠，可能会返回name值，上面有说明  
 Expr.find["ID"] = function( id, context ) {  
 if ( typeof context.getElementById !== "undefined" && documentIsHTML ) {  
 var node, i, elems,  
 elem = context.getElementById( id );  
  
 if ( elem ) {  
  
 // Verify the id attribute  
 node = elem.getAttributeNode("id");  
 if ( node && node.value === id ) {//准确的找到了id元素  
 return [ elem ];  
 }  
  
 // 回退到使用getElementByName方法  
 elems = context.getElementsByName( id );  
 i = 0;  
 while ( (elem = elems[i++]) ) {  
 node = elem.getAttributeNode("id");  
 if ( node && node.value === id ) {  
 return [ elem ];  
 }  
 }  
 }  
  
 return [];  
 }  
 };  
 }  
  
 // Tag  
 Expr.find["TAG"] = support.getElementsByTagName ?  
 function( tag, context ) {  
 if ( typeof context.getElementsByTagName !== "undefined" ) {  
 return context.getElementsByTagName( tag );  
  
 // DocumentFragment nodes 没有gEBTN  
 } else if ( support.qsa ) {  
 return context.querySelectorAll( tag );  
 }  
 } :  
  
 function( tag, context ) {  
 var elem,  
 tmp = [],  
 i = 0,  
 // 出于巧合，documentFragment的nodes有gEBTN方法  
 results = context.getElementsByTagName( tag );  
  
 // Filter out possible comments  
 if ( tag === "\*" ) {  
 while ( (elem = results[i++]) ) {  
 if ( elem.nodeType === 1 ) {  
 tmp.push( elem );  
 }  
 }  
  
 return tmp;  
 }  
 return results;  
 };  
  
 // Class  
 // getElementsByClassName htmlDOM  
 // 如support.getElementsByClassName == true，返回function( className, context )  
 Expr.find["CLASS"] = support.getElementsByClassName && function( className, context ) {  
 if ( typeof context.getElementsByClassName !== "undefined" && documentIsHTML ) {  
 return context.getElementsByClassName( className );  
 }  
 };  
  
 /\* QSA/matchesSelector  
 针对具体有问题的浏览器，提出解决办法  
 ---------------------------------------------------------------------- \*/  
  
 // 匹配选择器(:active) 实际为true时，却返回false(IE9/Opera 11.5)  
 rbuggyMatches = [];  
  
 // 匹配选择器(:focus) 实际为true时，却返回false (Chrome 21)  
 // 这个bug未修复是因为 IE8/9在iframe中访问document.activeElement会抛出错误  
 // 未修复这个bug是为了不让ie抛出错误  
 // See https://bugs.jquery.com/ticket/13378  
 rbuggyQSA = [];  
 // document.querySelectorAll:返回与指定的选择器组匹配的文档中的元素列表 。返回的对象是 NodeList 。  
 if ( (support.qsa = rnative.test( document.querySelectorAll )) ) {  
 // 创建QSA正则表达式，策略采用来源 Diego Perini  
 // el = document.createElement("fieldset");  
 assert(function( el ) {  
 // Select 元素设置为空字符串  
 // 解决https://bugs.jquery.com/ticket/12359问题  
 // 下面产生的html代码为：  
 // <a id="a"></a>  
 // <select id="a-  
 // \" msallowcapture="">  
 // <option selected=""></option>  
 // </select>  
 docElem.appendChild( el ).innerHTML = "<a id='" + expando + "'></a>" +  
 "<select id='" + expando + "-\r\\' msallowcapture=''>" +  
 "<option selected=''></option></select>";  
  
 // Support: IE8, Opera 11-12.16  
 // ^= , $= , \*=选择符后空字符串，应无内容选出  
 // https://msdn.microsoft.com/en-us/library/ie/hh465388.aspx#attribute\_section  
 if ( el.querySelectorAll("[msallowcapture^='']").length ) {  
 rbuggyQSA.push( "[\*^$]=" + whitespace + "\*(?:''|\"\")" );  
 }  
  
 // Support: IE8  
 // IEB对Boolean特性和值处理不正确  
 if ( !el.querySelectorAll("[selected]").length ) {  
 rbuggyQSA.push( "\\[" + whitespace + "\*(?:value|" + booleans + ")" );  
 }  
  
 // Support: Chrome<29, Android<4.4, Safari<7.0+, iOS<7.0+, PhantomJS<1.9.8+  
 if ( !el.querySelectorAll( "[id~=" + expando + "-]" ).length ) {  
 rbuggyQSA.push("~=");  
 }  
  
 // Webkit/Opera - :checked应该返回选中的元素，IE抛出异常  
 // http://www.w3.org/TR/2011/REC-css3-selectors-20110929/#checked  
 if ( !el.querySelectorAll(":checked").length ) {  
 rbuggyQSA.push(":checked");  
 }  
  
 // Support: Safari 8+, iOS 8+  
 // 类似的span#id + ok选择器返回结果不正确  
 // https://bugs.webkit.org/show\_bug.cgi?id=136851  
 if ( !el.querySelectorAll( "a#" + expando + "+\*" ).length ) {  
 rbuggyQSA.push(".#.+[+~]");  
 }  
 });  
  
 assert(function( el ) {  
 el.innerHTML = "<a href='' disabled='disabled'></a>" +  
 "<select disabled='disabled'><option/></select>";  
  
 // Support: Windows 8 Native Apps  
 // type和name特性，在.innerHTML使用时受限  
 var input = document.createElement("input");  
 input.setAttribute( "type", "hidden" );  
 el.appendChild( input ).setAttribute( "name", "D" );  
  
 // Support: IE8  
 // Enforce case-sensitivity of name attribute  
 if ( el.querySelectorAll("[name=d]").length ) {  
 rbuggyQSA.push( "name" + whitespace + "\*[\*^$|!~]?=" );  
 }  
  
 // FF 3.5 - :enabled/:disabled and hidden elements (hidden elements are still enabled)  
 // IE8 throws error here and will not see later tests  
 if ( el.querySelectorAll(":enabled").length !== 2 ) {  
 rbuggyQSA.push( ":enabled", ":disabled" );  
 }  
  
 // Support: IE9-11+  
 // IE's :disabled selector does not pick up the children of disabled fieldsets  
 docElem.appendChild( el ).disabled = true;  
 if ( el.querySelectorAll(":disabled").length !== 2 ) {  
 rbuggyQSA.push( ":enabled", ":disabled" );  
 }  
  
 // Opera 10-11 does not throw on post-comma invalid pseudos  
 el.querySelectorAll("\*,:x");  
 rbuggyQSA.push(",.\*:");  
 });  
 }  
 // el.matches（selectorString）:el能否被selectorString选择  
 // 几个浏览器以非标准名称实现了这个前缀matchesSelector()。  
 // 以下内容应足以满足大多数（如果不是全部）（即IE9 +支持）  
 if ( (support.matchesSelector = rnative.test( (matches = docElem.matches ||  
 docElem.webkitMatchesSelector ||  
 docElem.mozMatchesSelector ||  
 docElem.oMatchesSelector ||  
 docElem.msMatchesSelector) )) ) {  
  
 assert(function( el ) {  
 // 看能否在为挂在文档树中的node使用matches方法(IE9)  
 support.disconnectedMatch = matches.call( el, "\*" );  
  
 // 这段代码应抛出异常，但Gecko会返回false  
 matches.call( el, "[s!='']:x" );  
 rbuggyMatches.push( "!=", pseudos );  
 });  
 }  
  
 rbuggyQSA = rbuggyQSA.length && new RegExp( rbuggyQSA.join("|") );  
 rbuggyMatches = rbuggyMatches.length && new RegExp( rbuggyMatches.join("|") );  
  
 /\* Contains  
 ---------------------------------------------------------------------- \*/  
 hasCompare = rnative.test( docElem.compareDocumentPosition );  
  
 // 元素是否包含另一个  
 // contais：判断一个元素是否为指定元素后代  
 contains = hasCompare || rnative.test( docElem.contains ) ?  
 function( a, b ) {  
 // 如a是DOM根节点，返回顶层document对象  
 var adown = a.nodeType === 9 ? a.documentElement : a,  
 bup = b && b.parentNode;  
 // 如a包含b，则b.parentNode为a或a的子  
 return a === bup || !!( bup && bup.nodeType === 1 && (  
 adown.contains ? // 如contais方法存在，使用contains判断  
 adown.contains( bup ) :  
 // a.compareDocumentPosition( bup ) ==16表示a为bup的外部节点，如前2个&&为真，返回16  
 a.compareDocumentPosition && a.compareDocumentPosition( bup ) & 16  
 ));  
 } :  
 // 不存在contains和compareDocumentPosition方法，使用parentNode一层层比较  
 function( a, b ) {  
 if ( b ) {  
 while ( (b = b.parentNode) ) {  
 if ( b === a ) {  
 return true;  
 }  
 }  
 }  
 return false;  
 };  
  
 /\* Sorting  
 ---------------------------------------------------------------------- \*/  
  
 // document顺序  
 sortOrder = hasCompare ?  
 function( a, b ) {  
  
 if ( a === b ) {  
 hasDuplicate = true;  
 return 0;  
 }  
  
 // 如a，b只有一个有compareDocumentPosition方法，返回true-false：1或false-true：-1  
 var compare = !a.compareDocumentPosition - !b.compareDocumentPosition;  
 if ( compare ) {  
 return compare;  
 }  
  
 // 如a,b属于同一个文档，计算输入位置  
 compare = ( a.ownerDocument || a ) === ( b.ownerDocument || b ) ?  
 a.compareDocumentPosition( b ) :  
  
 // 否则无关系，不属于一个文档  
 1;  
  
 // 处理无关系两个节点关系  
 // a,b无关系，a.compareDocumentPostion(b)===1  
 // compare & 1 --true  
 // b.compareDocumentPosition( a ) === compare --true  
 // support.sortDetached,浏览器无bug时，返回true，有bug时返回false  
 if ( compare & 1 ||  
 (!support.sortDetached && b.compareDocumentPosition( a ) === compare) ) {  
  
 // Choose the first element that is related to our preferred document  
 if ( a === document || a.ownerDocument === preferredDoc && contains(preferredDoc, a) ) {  
 return -1;  
 }  
 if ( b === document || b.ownerDocument === preferredDoc && contains(preferredDoc, b) ) {  
 return 1;  
 }  
  
 // Maintain original order  
 return sortInput ?  
 ( indexOf( sortInput, a ) - indexOf( sortInput, b ) ) :  
 0;  
 }  
  
 return compare & 4 ? -1 : 1;  
 } :  
 function( a, b ) { //不存在compareDocumentPosition方法  
  
 if ( a === b ) {  
 hasDuplicate = true;  
 return 0;  
 }  
  
 var cur,  
 i = 0,  
 aup = a.parentNode,  
 bup = b.parentNode,  
 ap = [ a ],  
 bp = [ b ];  
  
 // 无父节点的不是documents节点，就是为挂在在dom树上的节点  
 if ( !aup || !bup ) {  
 return a === document ? -1 :  
 b === document ? 1 :  
 aup ? -1 :  
 bup ? 1 :  
 sortInput ?  
 ( indexOf( sortInput, a ) - indexOf( sortInput, b ) ) :  
 0;  
  
 // 如节点为同辈（父节点相同），快速检查  
 } else if ( aup === bup ) {  
 return siblingCheck( a, b );  
 }  
  
 // 否则需要全部祖先做比较  
 cur = a;  
 while ( (cur = cur.parentNode) ) {  
 ap.unshift( cur );  
 }  
 cur = b;  
 while ( (cur = cur.parentNode) ) {  
 bp.unshift( cur );  
 }  
  
 // 选择相同的祖先元素，只要dom树上元素一定存在  
 // 通过判断相同祖先的子元素，判断元素先后  
 while ( ap[i] === bp[i] ) {  
 i++;  
 }  
  
 return i ?  
 // Do a sibling check if the nodes have a common ancestor  
 siblingCheck( ap[i], bp[i] ) :  
  
 // Otherwise nodes in our document sort first  
 ap[i] === preferredDoc ? -1 :  
 bp[i] === preferredDoc ? 1 :  
 0;  
 };  
  
 return document;  
};

### 小结

* + - 1. 反观setDocument函数，目的是针对node设置document
      2. 如node为window.document直接返回，如不是
      3. 参看各方法的支持情况，添加到support的各个属性上
      4. 根据support不同，设置Expr.find和Expr.filter方法
      5. 根据QSAbug，将解决办法push到rbuggyQSA、rbuggyMatches数据
      6. 定义contains和sortOrder函数
         1. 问题：
         2. 1. rbuggyQSA、rbuggyMatches的push进的内容，为何这么push，怎么用

### 注意

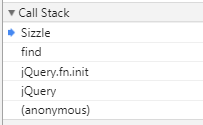
* + - 1. Line 2736，对setDocument进行了调用，初始化默认文档;
      2. 针对$(".c2 , span:first,.c1,input[type = button]")选择器，Sizzle函数参数分别为，selector=.c2 , span:first,.c1,input[type = button]；context=document，results=[prevObject: jQuery.fn.init(1)]，通过调用栈，可以查看完整的调用信息，通过断点，可以查看参数



## function Sizzle() ---(754)

### 概述

* + - 1. 此函数为选择器引擎的入口函数，随便写一个选择器$("div:first p,a")，在L755行打上断点，可以看到运行到此行时的调用栈为：Sizzle函数之前是，jQuery.find(selector, self[ i ], ret)，源码L2795, jQuery.find = Sizzle;而var Sizzle=(function(window){})(window)内部返回的是函数Sizzle



* + - 1. Sizzle函数

首先判断selector是否为单标签选择器，如$("#id"),$(".class"),$("span")等，如果是直接使用get\*By\*方法

接着，如果浏览器支持querySelectorAll，则优先使用querySelectorAll方法进行查询，查询到直接返回results，

最后，不符合上述情况的，传给select处理

* + - 1. 注意：

可根据源码逐行解析，看到具体函数跳到对应小结

### createCache(893)

* + - 1. arr.push返回添加数据之后数据长度，因此直接可以使用arr.push()>50来进行比较
      2. delete使用
         1. 对象属性删除（delete obj.name）
         2. 直接用delelte删除不了变量

/\*\*  
 \* 创建有限大小的键值缓存  
 \* @returns {function(string, object)} 返回一个对象，且数据存储在对象本身的，键为key + " "  
 \* 如cahce大于Expr.cachLength，删除最早加入数组的元素

\* 存储值时，var tokenCache = createCache()(key, value)

\*/  
function createCache() {  
 var keys = [];  
  
 function cache( key, value ) {  
 // 使用 (key + " ") 来避免与本地原型属性冲突(see Issue #157)  
 // Expr.cacheLength默认50，可以由使用者设置  
 // 如keys大于cache容量，将最旧的元素删除  
 if ( keys.push( key + " " ) > Expr.cacheLength ) {  
 // Only keep the most recent entries  
 delete cache[ keys.shift() ];  
 }  
 return (cache[ key + " " ] = value);  
 }  
 return cache;  
}

### Sizzle.tokenize(2170)

* + - 1. 主要作用是将selector表达式（$(“div,a>p”)中的div,a>p即为selector表达式）转换为group数组
      2. 关键是理解while ( soFar )循环，针对输入的selector会返回groups数组
      3. 源码逐行解析

/\*\*  
 \* 主要是用来转换selector  
 \* 如$("#div, span>a,.c1:first,input[type = button]")等  
 \* @type {Sizzle.tokenize}  
 \*/  
tokenize = Sizzle.tokenize = function( selector, parseOnly ) {  
 var matched, match, tokens, type,  
 soFar, groups, preFilters,  
 //只是检查tokenCache中有selector么  
 // tokenCache要返回数组，jquery返回groups，用slice(0)返回整个数组  
 cached = tokenCache[ selector + " " ];  
 if ( cached ) {  
 return parseOnly ? 0 : cached.slice( 0 );  
 }  
 soFar = selector;  
 groups = [];  
 preFilters = Expr.preFilter; // ATTR,CHILD,PSEUDO  
 // 将selector数据全解析完放入groups中  
 while ( soFar ) {//假设，soFar = selector = "#div, span>a,.c1:first,input[type = button]"  
  
 // 首先寻找逗号  
 // matched：控制循环  
 // 第1次循环，matched，match为undefined，创建groups这个二维数组，里面每项是tokens数组  
 // 第2次循环，soFar = ", span>a,.c1:first,input[type = button]"，此时matched=#div,但match为true  
 // rcomma.exec( soFar ):匹配以" ,"开头的表达式  
 if ( !matched || (match = rcomma.exec( soFar )) ) {  
 if ( match ) {  
 // 将匹配到的match数组从soFar中删除  
 // ||soFar，当两个逗号没有内容， soFar.slice( match[0].length )返回""  
 // 第2次循环： soFar = ", span>a,.c1:first,input[type = button]"  
 // 去除后变为soFar = "span>a,.c1:first,input[type = button]"  
 soFar = soFar.slice( match[0].length ) || soFar;  
 }  
 // 熟悉创建二维数组方式，后面可以直接往tokens中push  
 groups.push( (tokens = []) );  
 }  
 matched = false;  
 // 选择符  
 // 第3次循环，soFar=">a,.c1:first,input[type = button]"，进入此循环  
 // tokens = [value:'>',type:'>']  
 if ( (match = rcombinators.exec( soFar )) ) {  
 matched = match.shift();  
 tokens.push({  
 value: matched,  
 // Cast descendant combinators to space  
 type: match[0].replace( rtrim, " " )  
 });  
 soFar = soFar.slice( matched.length );  
 }  
  
 // 只是利用了Expr.filter 中的key，此key表明了sizzle引擎能处理是哪些类型  
 // matchExpr为匹配ID，TAG，CLASS，ATTR等的正则表达式  
 // preFilters为ATTR，CHILD，PSEUDO的过滤器函数  
 // 利用正则表达式看能匹配出哪个selector  
 // 第1次循环找出#div，matched = #div，故tokens1 = [value:'#div',type:'ID',matches:match]，groups= [tokens1]  
 // 通过 soFar.slice( matched.length )，去除#div后，soFar = ", span>a,.c1:first,input[type = button]"  
 for ( type in Expr.filter ) {  
 if ( (match = matchExpr[ type ].exec( soFar )) && (!preFilters[ type ] ||  
 (match = preFilters[ type ]( match ))) ) {  
 matched = match.shift();//获取匹配项  
 tokens.push({  
 value: matched,  
 type: type,  
 matches: match  
 });  
 soFar = soFar.slice( matched.length );//去除匹配项后的长度  
 }  
 }  
 // match中无数据，退出循环  
 if ( !matched ) {  
 break;  
 }  
 }  
  
 // parseOnly表示只是解析，返回soFar长度  
 // 否则返回tokens或抛出异常  
 return parseOnly ?  
 soFar.length :  
 soFar ?  
 Sizzle.error( selector ) :  
 // 缓存tokens  
 tokenCache( selector, groups ).slice( 0 );  
};

### function toSelector( tokens ) (2237)

* + - 1. tokenize的逆操作，将tokens拼接为selector

/\*\*  
 \* 获取tokens中对象的value值，并拼接在一起  
 \* @param tokens  
 \* @return {string}  
 \*/  
function toSelector( tokens ) {  
 var i = 0,  
 len = tokens.length,  
 selector = "";  
 for ( ; i < len; i++ ) {  
 selector += tokens[i].value;  
 }  
 return selector;  
}

### 源码逐行解析

// 引擎的主要入口函数  
// jquery.find未传递seed，并不代表Sizzle中的if(!seed)无意义，并不是seed一直是undefined  
// 此函数并不只是在jquery.find中调用一次，当selector复杂时，可能会被后面complie内部再次调用，  
// 如Line2329，multipleContexts中又调用Sizzle函数  
function Sizzle( selector, context, results, seed ) {  
 var m, i, elem, nid, match, groups, newSelector,  
 newContext = context && context.ownerDocument,  
  
 // nodetype=9为Document  
 nodeType = context ? context.nodeType : 9;  
  
 results = results || [];  
  
 // 如selector不是Element(1),DocumentFragment(11)返回results  
 if ( typeof selector !== "string" || !selector ||  
 nodeType !== 1 && nodeType !== 9 && nodeType !== 11 ) {  
  
 return results;  
 }  
  
 // 尝试在HTML文档中快捷地查找操作（而不是过滤器）  
 if ( !seed ) { //如果外界没有指定初始集合seed了。  
 // 如context不是document调用setDocument函数设置,因为传入window对象，document为window上属性  
 // preferredDoc = window.document  
 // ownerDocument:返回当前节点的顶层的document对象  
 if ( ( context ? context.ownerDocument || context : preferredDoc ) !== document ) {  
 setDocument( context );  
 }  
 context = context || document;  
 // !isXML( document )的为documentHtml  
 if ( documentIsHTML ) {  
 // 如selector足够简单（单标签选择器，如$("#id"),$(".class"),$("span")等），使用get\*By\*的DOM方法  
 // DocumentFragment不使用，因为DocumentFragment没有这些方法  
 if ( nodeType !== 11 && (match = rquickExpr.exec( selector )) ) {  
  
 // match[1]，match[2]，match[3]分别为id，元素，class选择器  
 if ( (m = match[1]) ) {  
  
 // Document context  
 if ( nodeType === 9 ) {  
 if ( (elem = context.getElementById( m )) ) {  
  
 // Support: IE, Opera, Webkit  
 // IE，Opera，Webkit某些版本有这个bug  
 // getElementById 能根据name匹配而不是id  
 if ( elem.id === m ) {  
 results.push( elem );  
 return results;  
 }  
 } else {  
 return results;  
 }  
  
 // Element context  
 } else {  
  
 // Support: IE, Opera, Webkit  
 // getElementById 能根据name匹配而不是id  
 // 不是document，利用顶层根元素获取m，并且保证context有elem  
 if ( newContext && (elem = newContext.getElementById( m )) &&  
 contains( context, elem ) &&  
 elem.id === m ) {  
  
 results.push( elem );  
 return results;  
 }  
 }  
  
 // Type selector  
 } else if ( match[2] ) {  
 push.apply( results, context.getElementsByTagName( selector ) );  
 return results;  
  
 // Class selector  
 } else if ( (m = match[3]) && support.getElementsByClassName &&  
 context.getElementsByClassName ) {  
  
 push.apply( results, context.getElementsByClassName( m ) );  
 return results;  
 }  
 }  
  
 // 利用querySelectorAll优势  
 // support.qsa = rnative.test( document.querySelectorAll )  
 // compilerCache[ selector + " " ] :缓存selector  
 // selector未缓存过，并未出现rbuggyQSA中的问题  
 if ( support.qsa &&  
 !compilerCache[ selector + " " ] &&  
 (!rbuggyQSA || !rbuggyQSA.test( selector )) ) {  
  
 if ( nodeType !== 1 ) {  
 newContext = context;  
 newSelector = selector;  
  
 // qSA会检查外部Element context，这并不是我们想要的  
 // Support: IE <=8  
 // 解决IE8下一个bug（Andrew Dupont提供的方法）  
 // 不理解bug起因，难理解bug修复？？？？？？？？？  
 } else if ( context.nodeName.toLowerCase() !== "object" ) {  
  
 // 捕获context的 ID, 如需要首先设置它  
 if ( (nid = context.getAttribute( "id" )) ) {  
 nid = nid.replace( rcssescape, fcssescape );  
 } else {  
 context.setAttribute( "id", (nid = expando) );  
 }  
  
 // 获取selector分组  
 groups = tokenize( selector );  
 i = groups.length;  
 while ( i-- ) {  
 groups[i] = "#" + nid + " " + toSelector( groups[i] );  
 }  
 newSelector = groups.join( "," );  
  
 // Expand context for sibling selectors  
 newContext = rsibling.test( selector ) && testContext( context.parentNode ) ||  
 context;  
 }  
  
 if ( newSelector ) {  
 try {  
 push.apply( results,  
 newContext.querySelectorAll( newSelector )  
 );  
 return results;  
 } catch ( qsaError ) { // 处理qsa问题  
 } finally {  
 if ( nid === expando ) {  
 context.removeAttribute( "id" );  
 }  
 }  
 }  
 }  
 }  
 }  
  
 // selector.replace( rtrim, "$1" )删除selector两端空格  
 // rtrim：匹配字符串两端空格，分组$1为不包含两端空格的内容  
 return select( selector.replace( rtrim, "$1" ), context, results, seed );  
}

## Sizzle.select = function( selector, context, results, seed )---(2656)

### 概述

因目前新浏览器大多都支持querySelectorAll，写较复杂selector不容易理解源码过程，故将L711-L882注释掉，让选择器直接进入select函数

此函数首先，判断selector中是否有分组（$(“div,a”)有逗号的表示有分组），如无分组则针对selector第一个为id，第二个为关系选择器(> + ~ )缩小文档查找范围，针对selector第一个不是关系选择器，且整个选择器无sizzle自定义伪类

最后，处理不了的传给compile生成编译函数再处理

注意：select函数通过处理特殊情况，获得加快后面编译函数运行的context或seed，进入complie产生编译函数

### 源码逐行解析

/\*\*  
 \* Sizzle()解析不了传给此函数，此函数首先处理无分组selector的某些情况  
 \* 然后传给Sizzle.compile解析  
 \* @param {String|Function} selector或由Sizzle.compile预编译的selector函数  
 \* @param {Element} context  
 \* @param {Array} [results]  
 \* @param {Array} [seed] A set of elements to match against  
 \*/  
select = Sizzle.select = function( selector, context, results, seed ) {  
 var i, tokens, token, type, find,  
 compiled = typeof selector === "function" && selector,  
 match = !seed && tokenize( (selector = compiled.selector || selector) );  
  
 results = results || []; //  
  
 // match中无分组(无逗号分隔情况下)，在编译前（执行complie函数）缩小范围  
 if ( match.length === 1 ) {  
  
 // 缩小文档查找范围，selector第一个为id，第二个为关系选择器(> + ~ )  
 // 对于$("#div > .p > a +b")，通过token[0]的id选择器，将context从document缩小为div#div.p  
 tokens = match[0] = match[0].slice( 0 );  
 if ( tokens.length > 2 && (token = tokens[0]).type === "ID" &&  
 context.nodeType === 9 && documentIsHTML && Expr.relative[ tokens[1].type ] ) {  
  
 context = ( Expr.find["ID"]( token.matches[0].replace(runescape, funescape), context ) || [] )[0];  
 if ( !context ) {  
 return results;  
  
 // 预编译matcher需要验证祖先，所以context提高为parentNode  
 } else if ( compiled ) {  
 context = context.parentNode;  
 }  
 // 去掉第一个token，即id  
 selector = selector.slice( tokens.shift().value.length );  
 }  
  
 // 缩小备选种子范围,从右向左匹配  
 // 第一个token是[>+~]关系选择器，或整个选择器中有sizzle自定义伪类,i为0，不进行循环  
 // 故，处理的是，第一个token不是关系选择器，且整个选择器无sizzle自定义伪类  
 i = matchExpr["needsContext"].test( selector ) ? 0 : tokens.length;  
 // 如对于 $("div > .p > a + b ")，第一次tokens[7]为b这个group（通过tokenize解析过的）  
 // 循环主要是处理$("div > .p > a +b:first-child ")情况，将:first-child跳过，去查询b  
 while ( i-- ) {  
 token = tokens[i];  
  
 // 遇到关系选择器则终止循环  
 if ( Expr.relative[ (type = token.type) ] ) {  
 break;  
 }  
 // 处理type为ID/CLASS/TAG的token,这些方法在setDocument中定义  
 if ( (find = Expr.find[ type ]) ) {  
 // find是Expr.find返回的函数，形如function( tag, context )  
 // 如查询到b，则seed中有值，否则为0，最后一个元素查询不到则不会有result  
 if ( (seed = find(  
 token.matches[0].replace( runescape, funescape ),  
 // rsibling = /[+~]/,  
 // aa && bb || cc ===>(aa && bb) || cc  
 // testContext判断是否为有效的sizzle上下文  
 rsibling.test( tokens[0].type ) && testContext( context.parentNode ) || context  
 )) ) {  
  
 // 如seed为空，或无tokens，直接返回  
 // tokens删除i这个元素  
 tokens.splice( i, 1 );  
 selector = seed.length && toSelector( tokens ); // 将tokens拼接为selector  
 // seed=0表示未找到b元素，故results为空  
 if ( !selector ) {  
 push.apply( results, seed );  
 return results;  
 }  
  
 break;  
 }  
 }  
 }  
 }  
  
 // 如编译函数不存在，会调用complie产生一个编译函数，因为Sizzle函数并不只是调用一次   
 // compiled存在，直接运行compiled(seed,context....)  
 // compiled为false，运行compile生成cache,cache为最终匹配函数，运行cache(seed,context....)  
 ( compiled || compile( selector, match ) )(  
 seed,  
 context,  
 !documentIsHTML,  
 results,  
 !context || rsibling.test( selector ) && testContext( context.parentNode ) || context  
 );  
 return results;  
};

**4.5.1 testContext( context ) ---(1804)**

/\*\*  
 \* 检查节点有效性作为Sizzle上下文

\* ？？？？为何这样就是Sizzle有效上下文

\* @param {Element|Object=} context  
 \* @returns {Element|Object|Boolean} The input node if acceptable, otherwise a falsy value  
 \*/  
function testContext( context ) {  
 return context && typeof context.getElementsByTagName !== "undefined" && context;  
}

## Sizzle.compile = function( selector, match )---(2617)

### 概述

* + - 1. 此函数主要是根据match(match即是selector通过tokenize解析后的groups数组)
      2. L2629这个while循环为i--，则表明selector是使用从右到左的解析方式生成编译函数的，以每个逗号为分组标志，循环解析每个分组
      3. 主要流程是：

首先，循环match数组，利用matcherFromTokens将match数组转换为编译函数matcher，然后将无伪类的分组push到elementMatchers数组，将有伪类的数组push到setMatchers数组

其次，利用matcherFromGroupMatchers函数将elementMatchers与setMatchers合并为最终匹配函数返回；

* + - 1. 利用较为简单$("div:first p,.rex > a")来分析complie函数生成
      2. Complie函数将selector分为两种情况，有无伪类；伪类让sizzle选择器选择到内容，可以在html中写一些代码（之后不对html做改变），如

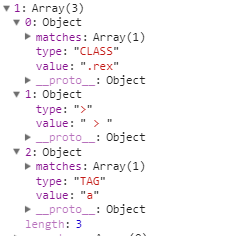
<div class="rex">  
　　<a>hello rextao</a>  
 <p class="group">group1</p>  
　　<p class="group">group2</p>  
　　<span class="group"></span>  
</div>  
<div class="aaa"></div>

### 源码逐行解析

/\*\*  
 \* 创建编译函数  
 \* 如对于$("div:first p,.rex > a"),第1分组push到setMatchers，第2分组push到elementMatchers  
 \* @type {Sizzle.compile}  
 \*/  
compile = Sizzle.compile = function( selector, match /\* Internal Use Only \*/ ) {  
 var i,  
 setMatchers = [],  
 elementMatchers = [],  
 // 查询当前选择器是否有被处理并保存在缓存compilerCache中  
 cached = compilerCache[ selector + " " ];  
  
 if ( !cached ) {  
 // 如果没有被词法解析先进行词法解析  
 if ( !match ) {  
 match = tokenize( selector );  
 }  
 i = match.length;  
 // 第1次迭代".rex > a"对应的tokens(match)数组  
 // 第2次迭代"div:first p" 对应的tokens(match)数组  
 while ( i-- ) {  
 // 根据tokens返回matcher函数  
 cached = matcherFromTokens( match[i] );  
 // 如果选择器中有伪类的选择器压入setMatchers，  
 // cached[expando]在生成匹配器函数(markFunction)的时候就判断是否有伪类而赋值了  
 if ( cached[ expando ] ) {  
 setMatchers.push( cached );  
 //普通选择器压入elementMatchers  
 } else {  
 elementMatchers.push( cached );  
 }  
 }  
 // 利用matcherFromGroupMatchers整合为终极匹配函数  
 cached = compilerCache( selector, matcherFromGroupMatchers( elementMatchers, setMatchers ) );  
  
 // 缓存selector  
 cached.selector = selector;  
 }  
 return cached;  
};

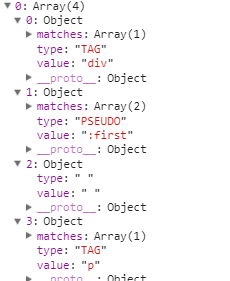
### 第1次迭代

由于选用$("div:first p,.rex > a")，complie第一次迭代，match[i]是".rex > a"对应的tokens(match)数组



### 第2次迭代

complie第2次迭代，match[i]是"div:first p "对应的tokens(match)数组，其中tokens.length= 4，因为div:first与p之间为子代关系，有一个关系选择器



## matcherFromTokens---(2448)

### 概述

\*针对一个块（不包含逗号分隔的选择器字符串）选择器生成匹配函数

\* 1. 针对不同类型的选择器产生不同的执行函数

\* 2. 若包含伪类，则返回setMatcher，否则返回的是elementMatcher，

\* 3. 通过鉴别matcher是否包含expando属性来区别是否有伪类

\* 4. 充当了selector“tokens”与Expr中定义的匹配方法的串联与纽带的作用

\* 5. 只是生成匹配函数，并未执行，最后统一执行

* + - 1. 注意：

此函数只是根据tokens生成匹配函数matcher，多为闭包，故函数并没有调用，在生成匹配函数期间，不容易理解某些函数为何这么比，为何传这个参数-。-

重点理解创建了何种编译函数

### 第1次迭代，处理无伪类的情况

* + - 1. 第1次迭代".rex > a"对应的tokens(match)数组，elementMatchers中的函数被层层包裹，Sizzle.compile中push到elementMatchers函数为

elementMatcher ([addCombinator(elementMatcher( [baseMatch,RexFuc] ), Obj>),aFuc]具体详见源码逐行解析(无伪类情况)

* + - 1. function addCombinator( matcher, combinator, base )---(2247)

/\*\*  
 \* 根据关系创建函数，如div > p，通过此函数处理>,"",+,~这些关系  
 \* 其中，matcher为div的匹配函数，combinator，为>,  
 \* 通过addCombinator联系起来返回一个具有关系的函数，即  
 \* addCombinator将“>”的处理和“div”关联起来  
 \* @param matcher  
 \* @param combinator Expr.relative中的值  
 \* @param base  
 \* @return {Function}  
 \*/  
function addCombinator( matcher, combinator, base ) {  
 var dir = combinator.dir,  
 // 默认Expr.relative并无next值  
 skip = combinator.next,  
 key = skip || dir,  
 // base为true时，>与""返回ture，+与~返回false  
 // checkNonElements字面意思检查是否为元素节点  
 checkNonElements = base && key === "parentNode",  
 doneName = done++; //判断是第几个关系选择器  
 // 是否为紧邻关系，  
 // 如div>p,表示div子代p（紧邻关系），div p（非紧邻），表示div内部所有p  
 return combinator.first ?  
 // 检查最接近的祖先/前元素  
 // 若关系选择器是>或+  
 function( elem, context, xml ) {  
 // 紧邻关系为何要迭代，有些浏览器会把节点之间换行符看作TextNode  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 return matcher( elem, context, xml );  
 }  
 }  
 return false;  
 } :  
  
 // 若关系选择器是""或~  
 function( elem, context, xml ) {  
 var oldCache, uniqueCache, outerCache,  
 // 用以区别不同的执行过程。  
 //每次执行addCombinator函数时，done变量都会加1，用以区别生成的不同的位置关系匹配函数。  
 newCache = [ dirruns, doneName ];  
  
 // 不能在xml节点上设置任意数据，所以，他们不会受益于缓存  
 if ( xml ) {  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 if ( matcher( elem, context, xml ) ) {  
 return true;  
 }  
 }  
 }  
 } else {  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 // 若elem节点的expando属性不存在，则赋予空对象  
 outerCache = elem[ expando ] || (elem[ expando ] = {});  
  
 // Support: IE <9 only  
 // 避免克隆attroperties (jQuery gh-1709)  
 uniqueCache = outerCache[ elem.uniqueID ] || (outerCache[ elem.uniqueID ] = {});  
  
 if ( skip && skip === elem.nodeName.toLowerCase() ) {  
 elem = elem[ dir ] || elem;  
 } else if ( (oldCache = uniqueCache[ key ]) &&  
 oldCache[ 0 ] === dirruns && oldCache[ 1 ] === doneName ) {  
  
 // 分配给newCache，所以，结果返回到以前元素  
 return (newCache[ 2 ] = oldCache[ 2 ]);  
 } else {  
 // 重新使用  
 uniqueCache[ key ] = newCache;  
  
 // 匹配则结束，未匹配继续检查  
 // newCache = [ dirruns, doneName ,matcher];  
 if ( (newCache[ 2 ] = matcher( elem, context, xml )) ) {  
 return true;  
 }  
 }  
 }  
 }  
 }  
 return false;  
 };  
}

* + - 1. function elementMatcher( matchers )---(2311)

/\*\*  
 \* 无伪类的匹配器匹配器，执行每个matchers函数返回true或false  
 \* @param matchers 每个元素都是非伪类的匹配器执行函数  
 \* @return {Function} 仅仅返回true或false，而非jquery对象  
 \*/  
function elementMatcher( matchers ) {  
 // matchers.length =1表示只有基础匹配函数，直接返回自己  
 return matchers.length > 1 ?  
 // 多匹配器情况下，要elem全部符合匹配器规则  
 function( elem, context, xml ) {  
 var i = matchers.length;  
 while ( i-- ) {  
 if ( !matchers[i]( elem, context, xml ) ) {  
 return false;  
 }  
 }  
 return true; // 匹配传入的elem节点是否满足要求，全部满足返回true  
 } :  
 matchers[0];  
}

* + - 1. 源码逐行解析(无伪类情况)

function matcherFromTokens( tokens ) {  
 var checkContext, matcher, j,  
 len = tokens.length,  
 leadingRelative = Expr.relative[ tokens[0].type ],//获取关系  
 //当获取不到关系给一个隐形关系parentNode  
 implicitRelative = leadingRelative || Expr.relative[" "],  
 i = leadingRelative ? 1 : 0,  
  
 // 基本匹配，确保元素从最顶层的context可找到  
 matchContext = addCombinator( function( elem ) {  
 return elem === checkContext;  
 }, implicitRelative, true ),  
 matchAnyContext = addCombinator( function( elem ) {  
 return indexOf( checkContext, elem ) > -1;  
 }, implicitRelative, true ),  
 // 确定元素在哪个context，也是最后一个执行的匹配器  
 // matchers[0]为一函数，通过判断checkContext.nodeType返回matchContext或matchAnyContext  
 // 假设这个基础match函数为baseMatch，故matchers=[baseMatch]  
 matchers = [ function( elem, context, xml ) {  
 var ret = ( !leadingRelative && ( xml || context !== outermostContext ) ) || (  
 (checkContext = context).nodeType ?  
 matchContext( elem, context, xml ) :  
 matchAnyContext( elem, context, xml ) );  
 // 避免挂在元素上(issue #299)  
 checkContext = null;  
 return ret;  
 } ];  
 // 无伪类时迭代的为".rex > a"对应的tokens数组,此循环为正序查找  
 // 第1次循环，处理.rex,如生成的matchers函数为RexFuc，循环结束matchers=[baseMatch,RexFuc]  
 // 第2次循环，处理>,此时的matcher为>对应的对象(L1664)，假设为 >,  
 // matchers=[baseMatch,RexFuc]经过elementMatcher包装，elementMatcher函数循环运行matchers，返回true或false  
 // addCombinator为关系("",+,>,~)处理函数，将Obj>与elementMatcher( matchers )进行包装，得到  
 // matchers = [addCombinator(elementMatcher( [baseMatch,RexFuc] ), Obj>) ]  
 // 第3次循环，处理a,如生成的matchers函数为aFuc，  
 // 循环结束matchers=[addCombinator(elementMatcher( [baseMatch,RexFuc] ), Obj>),  
 // aFuc]  
 // 因为addCombinator，elementMatcher此处不运行，  
 // addCombinator为何能处理关系，elementMatcher循环运行matchers又有何用等问题很难此时解决  
 // 最终返回给compile的函数又包裹了一层elementMathcer   
 for ( ; i < len; i++ ) {  
 // 判断tokens.type是否为关系选择器(>,"",+,~)  
 if ( (matcher = Expr.relative[ tokens[i].type ]) ) {  
 matchers = [ addCombinator(elementMatcher( matchers ), matcher) ];  
 } else {  
 // 根据tokens.type产生过滤函数  
 matcher = Expr.filter[ tokens[i].type ].apply( null, tokens[i].matches );  
  
 // 处理伪类(由markFunction创建的，才有expando属性)  
 // 只是返回函数并不运行  
 if ( matcher[ expando ] ) {  
 // 判断伪类后是否还有operator  
 j = ++i;  
 for ( ; j < len; j++ ) {  
 if ( Expr.relative[ tokens[j].type ] ) {  
 break;  
 }  
 }  
 return setMatcher(  
 i > 1 && elementMatcher( matchers ),  
 i > 1 && toSelector(  
 // 如当前位置之前的token是一个后代组合，插入隐含的\*  
 // 类似 "div :first"这样的选择器，传入setMatcher为div \*  
 tokens.slice( 0, i - 1 ).concat({ value: tokens[ i - 2 ].type === " " ? "\*" : "" })  
 ).replace( rtrim, "$1" ),  
 matcher,  
 i < j && matcherFromTokens( tokens.slice( i, j ) ),  
 // 从tokens的j元素选取，即伪类后面元素  
 j < len && matcherFromTokens( (tokens = tokens.slice( j )) ),  
 j < len && toSelector( tokens )  
 );  
 }  
 matchers.push( matcher );  
 }  
 }  
 // elementMatcher，即是循环matchers(elem, context, xml),返回true或false  
 return elementMatcher( matchers );  
}

### 第2次迭代，处理伪类的情况

function matcherFromTokens( tokens ) {  
 var checkContext, matcher, j,  
 len = tokens.length,  
 leadingRelative = Expr.relative[ tokens[0].type ],//获取关系  
 //当获取不到关系给一个隐形关系parentNode  
 implicitRelative = leadingRelative || Expr.relative[" "],  
 i = leadingRelative ? 1 : 0,  
  
 // 基本匹配，确保元素从最顶层的context可找到  
 matchContext = addCombinator( function( elem ) {  
 return elem === checkContext;  
 }, implicitRelative, true ),  
 matchAnyContext = addCombinator( function( elem ) {  
 return indexOf( checkContext, elem ) > -1;  
 }, implicitRelative, true ),  
 // 确定元素在哪个context，也是最后一个执行的匹配器  
 // matchers[0]为一函数，通过判断checkContext.nodeType返回matchContext或matchAnyContext  
 // 假设这个基础match函数为baseMatch，故matchers=[baseMatch]  
 matchers = [ function( elem, context, xml ) {  
 var ret = ( !leadingRelative && ( xml || context !== outermostContext ) ) || (  
 (checkContext = context).nodeType ?  
 matchContext( elem, context, xml ) :  
 matchAnyContext( elem, context, xml ) );  
 // 避免挂在元素上(issue #299)  
 checkContext = null;  
 return ret;  
 } ];  
 // 第2次循环，处理"div:first p"这个token数组时，len=4;  
 // i = 0时，处理div，生成matchers函数为funDiv，此时matchers为[baseMatch,funDiv]  
 for ( ; i < len; i++ ) {  
 // 如果有父子选择器（关系选择器）  
 if ( (matcher = Expr.relative[ tokens[i].type ]) ) {  
 matchers = [ addCombinator(elementMatcher( matchers ), matcher) ];  
 } else {  
 matcher = Expr.filter[ tokens[i].type ].apply( null, tokens[i].matches );  
  
 // 处理伪类(由markFunction创建的，才有expando属性)  
 // 只是返回函数并不运行  
 // 假设此时选择器$("div:first p")  
 // i = 0时，处理div，生成matchers函数为funDiv，此时matchers为[baseMatch,funDiv]  
 // i = 1时，处理:first，此时i=1，经过j =++i，i与j都为2，此时Expr.relative[ tokens[j].type，  
 // 表示的为""（div:first后的子代选择器），故为true，退出当前循环，返回setMatcher，参数分别为：  
 // 参数1：[baseMatch,funDiv]  
 // 参数2：:first的前置selector，即div，toSelector里面的替换  
 // 只是针对类似 "div :first"这样的选择器的特殊处理，并未改变tokens数组长度  
 // 参数3: :first的matcher函数  
 // 参数4: :false  
 // 参数5: :选取tokens数组伪类之后部分，此时伪类为i=1,故选取j=2以后部分，  
 // 即p元素，将p元素再传入matcherFromTokens进行解析得到一系列函数，  
 // 针对$("div:first p")情况，传入参数为[baseMatch,funP]  
 // 参数6：将tokens用toSelector拼接为selector  
 // setMatcher会返回一个markFunciton函数，这只是定义函数，等程序运行到setMatcher时再分析  
 // 伪类后面的选择器直接传入到setMatcher中，并将setMatcher返回的markFunction函数返回到complie中的setMatchers中  
 if ( matcher[ expando ] ) {  
 // 判断伪类后是否还有operator  
 j = ++i;  
 for ( ; j < len; j++ ) {  
 if ( Expr.relative[ tokens[j].type ] ) {  
 break;  
 }  
 }  
 return setMatcher(  
 i > 1 && elementMatcher( matchers ),  
 i > 1 && toSelector(  
 // 如当前位置之前的token是一个后代组合，插入隐含的\*  
 // 类似 "div :first"这样的选择器，传入setMatcher为div \*  
 tokens.slice( 0, i - 1 ).concat({ value: tokens[ i - 2 ].type === " " ? "\*" : "" })  
 ).replace( rtrim, "$1" ),  
 matcher,  
 i < j && matcherFromTokens( tokens.slice( i, j ) ),  
 // 从tokens的j元素选取，即伪类后面元素  
 j < len && matcherFromTokens( (tokens = tokens.slice( j )) ),  
 j < len && toSelector( tokens )  
 );  
 }  
 matchers.push( matcher );  
 }  
 }  
 // elementMatcher，即是循环matchers(elem, context, xml),返回true或false  
 return elementMatcher( matchers );  
}

### 两次迭代之后，将编译函数返回到compile中

当循环结束，matcherFromTokens分别产生了无伪类的编译函数elementMatchers和有伪类的编译函数setMatchers，L2639行，matcherFromGroupMatchers将上面两个编译函数组合为最终super匹配函数，并缓存在compilerCache中，将cached返回到Sizzle.select中

其中，matcherFromGroupMatchers，创建了一个superMatcher函数，通过判断selector中有伪类，返回markFunction( superMatcher )，无伪类返回superMatcher

最后，L2644行，将最终的匹配函数cached返回到Sizzle.select，开始L2716后面括号，传入参数开始运行这个superMatcher编译函数

## Sizzle.compile函数运行

Compile函数返回的cached是matcherFromGroupMatchers构造是superMatcher，故compile运行即首先运行superMatcher，即matcherFromGroupMatchers函数

### matcherFromGroupMatchers( elementMatchers, setMatchers )----（2506）

此函数运行分两大主要部分进行处理，L2529处理无伪类的selector情况，L2572处理有伪类的情况

通过L2599，将results结果去重后，返回到compile函数中并返回Sizzle引擎处理的最终结果

* + - 1. 源码逐行分析

/\*\*  
 \* 将不同的块选择器生成的最终执行函数，该函数还负责将最终结果过滤掉重复对象。  
 \* @param elementMatchers  
 \* @param setMatchers  
 \* @return {superMatcher}  
 \*/  
function matcherFromGroupMatchers( elementMatchers, setMatchers ) {  
 var bySet = setMatchers.length > 0,  
 // seed,context首次调用未改变，故seed为undefined，context为document  
 // xml:true，or false，是否为html  
 // result:select方法中查找结果集，sizzle函数不只允许1次，故首次调用results为[]  
 byElement = elementMatchers.length > 0,  
 superMatcher = function( seed, context, xml, results, outermost ) {  
 var elem, j, matcher,  
 matchedCount = 0,  
 i = "0",  
 unmatched = seed && [],  
 setMatched = [],  
 contextBackup = outermostContext,  
 // 必须总是具有seed或外部context  
 // 确定起始查找范围或是参数中传递过来的备选种子seed,或是搜索范围context的所有后代节点  
 // 如seed集合不存在，elems则是html中全部tag，通过一个个循环匹配elementMatchers函数  
 elems = seed || byElement && Expr.find["TAG"]( "\*", outermost ),  
 //使用整数dirruns当且仅当这是最外面的匹配  
 dirrunsUnique = (dirruns += contextBackup == null ? 1 : Math.random() || 0.1),  
 len = elems.length;  
  
 if ( outermost ) {  
 outermostContext = context === document || context || outermost;  
 }  
  
 // elementMatchers匹配成功的元素直接加入到result集合中  
 // 因为没有seed集合，elems为html全部tag，逐个匹配matcher函数，看哪个元素节点符合匹配函数  
 // elem循环次序，html，head，meta，title，body，div.rex等等  
 for ( ; i !== len && (elem = elems[i]) != null; i++ ) {  
 if ( byElement && elem ) {  
 j = 0;  
 if ( !context && elem.ownerDocument !== document ) {  
 setDocument( elem );  
 xml = !documentIsHTML;  
 }  
 // 执行elementMatchers函数  
 // 此时的elementMatchers为".rex > a"对应的编译函数，  
 // 因为被层层包裹为elementMatcher ([addCombinator(elementMatcher( [baseMatch,RexFuc] ), Obj>),aFuc]  
 // 实际elementMatchers.length = 1；  
 // (matcher = elementMatchers[j++])将elementMatcher返回函数赋给matcher  
 // matcher( elem, context || document, xml),循环执行matchers，运行（elementMatcher函数），详细参见function elementMatcher( matchers )  
 // 如匹配不到，matcher返回false，j=1，再循环一次，此时elementMatchers[1]为undefined，while循环为false，退出循环  
 // 故matchedCount--  
 // 如匹配到,matcher返回true,将elem加到results里面，然后退出循环，故此时!matcher为false  
 // 故matchedCount不在进行--操作  
 while ( (matcher = elementMatchers[j++]) ) {  
 if ( matcher( elem, context || document, xml) ) {  
 results.push( elem );  
 break;  
 }  
 }  
 if ( outermost ) {  
 dirruns = dirrunsUnique;  
 }  
 }  
  
 // 追踪未匹配元素  
 // 如selector存在伪类，则bySet>0,matchedCount  
 if ( bySet ) {  
 // 当有matcher函数返回为true时  
 if ( (elem = !matcher && elem) ) {  
 matchedCount--;  
 }  
  
 // Lengthen the array for every element, matched or not  
 if ( seed ) {  
 unmatched.push( elem );  
 }  
 }  
 }  
  
 // `i` 现在是上面访问元素的个数，把他与matchedCount加在一起，使matchedCount非负  
 // 如不存在伪类，bySet为false，故matchedCount==i，上面循环会找到全部elementMatchers元素  
 matchedCount += i;  
  
 // 对unmatched元素应用set过滤器  
 // NOTE: 如无未匹配元素，则跳过（如matchedCount ==i），  
 // 除非上述循环中因为无seed或无元素matchers造成未访问任何元素  
 if ( bySet && i !== matchedCount ) {  
 j = 0;  
 // 无seed时，unmatched为seed&&[]，故为undefined，setMatched为[]  
 while ( (matcher = setMatchers[j++]) ) {  
 // unmatched：为传入setMatcher的seed  
 // setMatched:为[]  
 // context:document  
 // xml :为true或false  
 matcher( unmatched, setMatched, context, xml );  
 }  
  
 if ( seed ) {  
 // Reintegrate element matches to eliminate the need for sorting  
 if ( matchedCount > 0 ) {  
 while ( i-- ) {  
 if ( !(unmatched[i] || setMatched[i]) ) {  
 setMatched[i] = pop.call( results );  
 }  
 }  
 }  
  
 // Discard index placeholder values to get only actual matches  
 setMatched = condense( setMatched );  
 }  
  
 // 将matches加入result结果集  
 push.apply( results, setMatched );  
  
 // 对结果去重和排序  
 // 例如$("div:first a,.rex > a")查询到的可能是同一个a标签，  
 // 这样results中有两个一样的a，因此通过Sizzle.uniqueSort将重复的去掉  
 if ( outermost && !seed && setMatched.length > 0 &&  
 ( matchedCount + setMatchers.length ) > 1 ) {  
  
 Sizzle.uniqueSort( results );  
 }  
 }  
  
 // 用嵌套匹配函数覆盖全局函数  
 if ( outermost ) {  
 dirruns = dirrunsUnique;  
 outermostContext = contextBackup;  
 }  
  
 return unmatched;  
 };  
  
 return bySet ?  
 markFunction( superMatcher ) :  
 superMatcher;  
}

### function elementMatcher( matchers ) ----（2311）

函数在L2537调用，此时因为无seed并且context为document，故调用时，elem实际为html页面全部的tag标签（上面已经介绍），当外围（L2529）每次循环时，elem从html，head，meta，title，body，div.rex逐步循环

此时matchers为".rex > a"的编译函数elementMatcher ([addCombinator(elementMatcher ( [baseMatch,RexFuc] ), Obj>),aFuc]，此while循环从后往前找，故首次循环查找的是aFunc，即在elem(html，head，meta，title，body，div.rex等)中查询a标签

如何查找a标签呢？利用matchers[i]( elem, context, xml )，即用L1753行的Expr.filter[“Tag”]，对于a标签，nodeName为a，elem.nodeName根据外围循环，分别为HTML，HEAD，META等，无匹配返回false，当matchers[i]( elem, context, xml )返回false，则直接返回false给matcherFromGroupMatchers函数

"TAG": function( nodeNameSelector ) {  
 var nodeName = nodeNameSelector.replace( runescape, funescape ).toLowerCase();  
 return nodeNameSelector === "\*" ?  
 function() { return true; } :  
 function( elem ) {  
 return elem.nodeName && elem.nodeName.toLowerCase() === nodeName;  
 };  
}

但对于本文的html代码，当i为4时，即为a标签，Expr.filter[“Tag”]返回true，elementMatcher则接着循环，判断matchers[0],即[addCombinator(elementMatcher ( [baseMatch,RexFuc] ), Obj>)，如下为addCombinator部分代码（详细见后面）

return combinator.first ?  
 function( elem, context, xml ) {  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 return matcher( elem, context, xml );  
 }  
 }  
 return false;  
 } :

{……………}

combinator为Obj>，即.rex > a关系选择器（>），匹配关系在L1664，combinator.first:true表示直接相邻关系，根据a.parentNode，判断是否元素节点，是则返回matcher函数运行结果，此时matcher为elementMatcher ( [baseMatch,RexFuc] )，又进入elementMatcher，继续运行，查找.rex是否存在，只是此时的elem不在是a，而是a.parentNode

小结，elementMatcher是通过外围不断循环传入的elem(html,head,meta等对象集合)，查询当前selector是否存在，如查询到a存在，则根据关系匹配符(>)，将elem按照规则转换a.parent，再次判断a.parent是否有.rex

//匹配关系  
relative: {  
 ">": { dir: "parentNode", first: true },  
 " ": { dir: "parentNode" },  
 "+": { dir: "previousSibling", first: true },  
 "~": { dir: "previousSibling" }  
},

* + - 1. elementMatcher代码

function elementMatcher( matchers ) {  
 return matchers.length > 1 ?  
 function( elem, context, xml ) {  
 var i = matchers.length;  
 while ( i-- ) {  
 if ( !matchers[i]( elem, context, xml ) ) {  
 return false;  
 }  
 }  
 return true;  
 } :  
 matchers[0];  
}

### addCombinator( matcher, combinator, base ) ---(L2247)

/\*\*  
 \* 根据关系创建函数，如div > p，通过此函数处理>,"",+,~这些关系  
 \* 其中，matcher为div的匹配函数，combinator，为>,  
 \* 通过addCombinator联系起来返回一个具有关系的函数，即  
 \* addCombinator将“>”的处理和“div”关联起来  
 \* @param matcher  
 \* @param combinator Expr.relative中的值  
 \* @param base  
 \* @return {Function}  
 \*/  
function addCombinator( matcher, combinator, base ) {  
 var dir = combinator.dir,  
 // 默认Expr.relative并无next值  
 skip = combinator.next,  
 key = skip || dir,  
 // base为true时，>与""返回ture，+与~返回false  
 // checkNonElements字面意思检查是否为元素节点  
 checkNonElements = base && key === "parentNode",  
 doneName = done++; //判断是第几个关系选择器  
 // 是否为紧邻关系，  
 // 如div>p,表示div子代p（紧邻关系），div p（非紧邻），表示div内部所有p  
 return combinator.first ?  
 // 检查最接近的祖先/前元素  
 // 若关系选择器是>或+  
 function( elem, context, xml ) {  
 // 紧邻关系为何要迭代，有些浏览器会把节点之间换行符看作TextNode  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 return matcher( elem, context, xml );  
 }  
 }  
 return false;  
 } :  
  
 // 若关系选择器是""或~  
 function( elem, context, xml ) {  
 var oldCache, uniqueCache, outerCache,  
 // 用以区别不同的执行过程。  
 //每次执行addCombinator函数时，done变量都会加1，用以区别生成的不同的位置关系匹配函数。  
 newCache = [ dirruns, doneName ];  
  
 // 不能在xml节点上设置任意数据，所以，他们不会受益于缓存  
 if ( xml ) {  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 if ( matcher( elem, context, xml ) ) {  
 return true;  
 }  
 }  
 }  
 } else {  
 while ( (elem = elem[ dir ]) ) {  
 if ( elem.nodeType === 1 || checkNonElements ) {  
 // 若elem节点的expando属性不存在，则赋予空对象  
 outerCache = elem[ expando ] || (elem[ expando ] = {});  
  
 // Support: IE <9 only  
 // 避免克隆attroperties (jQuery gh-1709)  
 uniqueCache = outerCache[ elem.uniqueID ] || (outerCache[ elem.uniqueID ] = {});  
  
 if ( skip && skip === elem.nodeName.toLowerCase() ) {  
 elem = elem[ dir ] || elem;  
 } else if ( (oldCache = uniqueCache[ key ]) &&  
 oldCache[ 0 ] === dirruns && oldCache[ 1 ] === doneName ) {  
  
 // 分配给newCache，所以，结果返回到以前元素  
 return (newCache[ 2 ] = oldCache[ 2 ]);  
 } else {  
 // 重新使用  
 uniqueCache[ key ] = newCache;  
  
 // 匹配则结束，未匹配继续检查  
 // newCache = [ dirruns, doneName ,matcher];  
 if ( (newCache[ 2 ] = matcher( elem, context, xml )) ) {  
 return true;  
 }  
 }  
 }  
 }  
 }  
 return false;  
 };  
}

### setMatcher( preFilter, selector, matcher, postFilter, postFinder, postSelector ) ---（L2355）

matcherFromGroupMatchers函数在，L2572处理selector中有伪类的情况，并在L2575调用matcher( unmatched, setMatched, context, xml )，unmatched为传入setMatcher闭包的seed，setMatched为[]，xml :为true或false，函数调用后进入setMatcher的return函数(L2362)中

* + - 1. multipleContexts( selector, contexts, results )-----(L2325)

主要是将有伪类的selector分组，如$("div:first p ")，此时selector为div，会将div再传入到Sizzle函数中查询，会得到div.rex和div.aaa

/\*\*  
 \* setMatcher中，处理setMatcher第二个参数，将selector再传入Sizzle中解析  
 \* 返回selector查询到的结果  
 \* @param selector  
 \* @param contexts  
 \* @param results  
 \* @return {\*}  
 \*/  
function multipleContexts( selector, contexts, results ) {  
 var i = 0,  
 len = contexts.length;  
 for ( ; i < len; i++ ) {  
 Sizzle( selector, contexts[i], results );  
 }  
 return results;  
}

setMatcher返回的markFunction的L2389，此时的matcher实为Expr.pseudos.first的matcher函数(L2109)，所有的关于位置处理的（:first,:last,:eq,:even等）都是调用了createPositionalPseudo函数

* + - 1. Expr.pseudos.first----(L2109)

"first": createPositionalPseudo(function() {  
 return [ 0 ];//返回数组第一个元素  
})

* + - 1. createPositionalPseudo( fn )-----(L1061)

.first情况最为简单，fn只是返回匹配数组的第一项即可，因此matchIndexex = [0],i=1;此时seed为div匹配到元素，即div.rex和div.aaa; matches为空，并将matches，seed进行修改后返回到setMatcher函数中

/\*\*  
 \* 返回一个在Expr.Pseudo使用的伪类位置函数  
 \* 会创建fn[expandoo] = true  
 \* 以Expr.pseudos.first为例，此返回[0];  
 \* @param {Function} fn  
 \*/  
function createPositionalPseudo( fn ) {  
 return markFunction(function( argument ) {  
 argument = +argument;  
 return markFunction(function( seed, matches ) {  
 var j,  
 // 对于Expr.pseudos.first,matchIndexes为[0]  
 matchIndexes = fn( [], seed.length, argument ),  
 i = matchIndexes.length;  
  
 // 匹配指定下标元素  
 while ( i-- ) {  
 if ( seed[ (j = matchIndexes[i]) ] ) {  
 seed[j] = !(matches[j] = seed[j]);  
 }  
 }  
 });  
 });  
}

因为!!seed与!!postFilter为false，因此程序运行到L2439，因为setMatcher函数开始将postFinder用setMatcher又包裹了一层，故当postFinder为真时，setMatcher又运行一遍去查询$(‘div:first p’)中的p，此时的context已经根据上述查询出的结果matcherOut改为div.rex

将最终的结果返回到matcherFromGroupMatchers的results集中

* + - 1. 源码逐行解析

/\*\*  
 \* 伪类分割器  
 \* 假如选择器$("div:first p")  
 \* @param preFilter [baseMatch,funDiv]  
 \* @param selector first的前置selector，即div  
 \* @param matcher first的matcher函数  
 \* @param postFilter false  
 \* @param postFinder 选取tokens数组伪类之后部分，传入参数为[baseMatch,funP]  
 \* @param postSelector 将tokens用toSelector拼接为selector  
 \* @return {\*}  
 \*/  
function setMatcher( preFilter, selector, matcher, postFilter, postFinder, postSelector ) {  
 if ( postFilter && !postFilter[ expando ] ) {  
 postFilter = setMatcher( postFilter );  
 }  
 // 后置匹配器postFinder存在，且无伪类，则将postFinder用setMatcher包装  
 if ( postFinder && !postFinder[ expando ] ) {  
 postFinder = setMatcher( postFinder, postSelector );  
 }  
 return markFunction(function( seed, results, context, xml ) {  
 var temp, i, elem,  
 preMap = [],  
 postMap = [],  
 preexisting = results.length,  
  
 // 从seed或context中获取初始化参数  
 // elems为匹配到的节点；  
 // multipleContexts主要是将伪类的前置选择器（div）代入Sizzle函数中查询  
 // 本例，elems查询到的为div.rex和div.aaa  
 elems = seed || multipleContexts( selector || "\*", context.nodeType ? [ context ] : context, [] ),  
  
 // 利用预处理器进行匹配，主要是保证seed结果同步  
 // 对于本例子，seed不存在，故matcherIn返回elems  
 matcherIn = preFilter && ( seed || !selector ) ?  
 condense( elems, preMap, preFilter, context, xml ) :  
 elems,  
 // 注意多个三元表达式连接时顺序；通过代码层级也可得知  
 // matcherOut = matcher ?（postFinder || (() ? [] :results)） ：results  
 matcherOut = matcher ?  
 // 如果存在postFinder，或filtered seed，non-seed postFilter，preexisting results  
 postFinder || ( seed ? preFilter : preexisting || postFilter ) ?  
  
 // 中间处理是必要的。  
 [] :  
  
 // 否则直接用results结果  
 results :  
 matcherIn;  
 // 针对$("div:first p")，此处matcher其实是:first的matcher  
 // matcher为Expr.pseudos.first,matcher中会根据内部内容修改matcherIn，matcherOut这些传入参数  
 // 因此matcher运行完，matcherIn = (2) [false, div.aaa], matcherOut = [div.rex]  
 if ( matcher ) {  
 matcher( matcherIn, matcherOut, context, xml );  
 }  
  
 // 此例postFilter为fasle  
 if ( postFilter ) {  
 temp = condense( matcherOut, postMap );  
 postFilter( temp, [], context, xml );  
  
 // Un-match failing elements by moving them back to matcherIn  
 i = temp.length;  
 while ( i-- ) {  
 if ( (elem = temp[i]) ) {  
 matcherOut[ postMap[i] ] = !(matcherIn[ postMap[i] ] = elem);  
 }  
 }  
 }  
  
 if ( seed ) {  
 if ( postFinder || preFilter ) {  
 if ( postFinder ) {  
 // Get the final matcherOut by condensing this intermediate into postFinder contexts  
 temp = [];  
 i = matcherOut.length;  
 while ( i-- ) {  
 if ( (elem = matcherOut[i]) ) {  
 // Restore matcherIn since elem is not yet a final match  
 temp.push( (matcherIn[i] = elem) );  
 }  
 }  
 postFinder( null, (matcherOut = []), temp, xml );  
 }  
  
 // Move matched elements from seed to results to keep them synchronized  
 i = matcherOut.length;  
 while ( i-- ) {  
 if ( (elem = matcherOut[i]) &&  
 (temp = postFinder ? indexOf( seed, elem ) : preMap[i]) > -1 ) {  
  
 seed[temp] = !(results[temp] = elem);  
 }  
 }  
 }  
  
 // 如果postFinder定义了， 则通过postFinder为结果集添加元素  
 } else {  
 matcherOut = condense(  
 matcherOut === results ?  
 matcherOut.splice( preexisting, matcherOut.length ) :  
 matcherOut  
 );  
 // postFinder存在，调用postFinder会接着运行setMatcher，因为postFinder被postFinder包裹  
 // 因为通过第一次setMatcher，匹配出matcherOut，即div:first是哪个元素，div.rex  
 // 相当于此时再次调用setMatcher函数，是将context改为div.rex去查询符合的p  
 if ( postFinder ) {  
 postFinder( null, results, matcherOut, xml );  
 } else {  
 push.apply( results, matcherOut );  
 }  
 }  
 });

## 处理Seed情况

select函数在L2665行，通过判断match.length === 1来缩减context和寻找seed，如有分组，则直接进入compile；

对于$(‘div a’)满足寻找seed条件的selector，L2686从selector最后一个元素(a)开始循环，会查询到seed集合a，并将原始的selector转为”div ”（注意div后有个子代选择器），这时将新的selector传入compile生成编译函数后compile运行

Compile运行时，因为seed集合的存在，L2517行elems = seed || byElement && Expr.find["TAG"]( "\*", outermost )的elems不再是html,head,body等全部页面对象集合，而是seed，缩减了查找范围

总之，select函数处理特殊情况的结果就是，缩减context和寻找seed

## Expr = Sizzle.selectors(1651)

### markFunction()

/\*\*  
 \* 标记一个函数为Sizzle特殊用途  
 \* expando = "sizzle" + 1 \* new Date()  
 \* fn.expando =true;  
 \* @param {Function} fn The function to mark  
 \*/  
function markFunction( fn ) {  
 fn[ expando ] = true;  
 return fn;  
}

### Expr. Filter---(1746)

filter: {  
 // 如下函数创建，都是在matcherFromTokens函数中，参数为tokens.matches提供,此tokens.matches为  
 // matchExpr[ type ].exec( selectors )的结果  
 // 函数运行到matcherFromGroupMatchers才能理解，matcherFromGroupMatchers中会调用  
 // 其中elem为外层循环的html、head、body等对象  
 // nodeNameSelector为selector，如div  
 // 如当循环运行到elem为head，则elem.nodeName = HEAD && head === div  
 "TAG": function( nodeNameSelector ) {  
 var nodeName = nodeNameSelector.replace( runescape, funescape ).toLowerCase();  
 return nodeNameSelector === "\*" ?  
 function() { return true; } :  
 function( elem ) { //html中nodeName会返回大写  
 return elem.nodeName && elem.nodeName.toLowerCase() === nodeName;  
 };  
 },  
 // 判断元素上有某个类  
 "CLASS": function( className ) {  
 var pattern = classCache[ className + " " ];  
  
 return pattern ||  
 (pattern = new RegExp( "(^|" + whitespace + ")" + className + "(" + whitespace + "|$)" )) &&  
 classCache( className, function( elem ) {  
 return pattern.test( typeof elem.className === "string" && elem.className || typeof elem.getAttribute !== "undefined" && elem.getAttribute("class") || "" );  
 });  
 },  
 // 对于 $("div[class='rex']")，根据matcherFromTokens生成matcher函数的  
 // matcher = Expr.filter[ tokens[i].type ].apply( null, tokens[i].matches );  
 // 可得知tokens.matches分别为class,=,rex,分别为参数name, operator, check  
 "ATTR": function( name, operator, check ) {  
 return function( elem ) { //返回一个attr匹配器  
 // 获取某个元素节点elem上属性为name的属性值  
 var result = Sizzle.attr( elem, name );  
 //看看属性值有木有！  
 if ( result == null ) {  
 // 属性不存在，是不等于任何值的，故判断operator是否为！=  
 return operator === "!=";  
 }  
 if ( !operator ) {  
 return true;  
 }  
  
 result += "";  
  
 return operator === "=" ? result === check :  
 operator === "!=" ? result !== check :  
 // 判断目标值是否在当前属性值的头部  
 operator === "^=" ? check && result.indexOf( check ) === 0 :  
 // lang\*=en 匹配这样 <html lang="xxxxenxxx">的节点，包含en子串就行  
 // 在result能查询到check  
 operator === "\*=" ? check && result.indexOf( check ) > -1 :  
 // 判断是否为结尾\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 operator === "$=" ? check && result.slice( -check.length ) === check :  
 // 选择包含check这个词的，  
 operator === "~=" ? ( " " + result.replace( rwhitespace, " " ) + " " ).indexOf( check ) > -1 :  
 // 匹配这样 <html lang="en-US">的节点  
 operator === "|=" ? result === check || result.slice( 0, check.length + 1 ) === check + "-" :  
 false;  
 };  
 },  
 // 对于$("div:nth-child(2)")，tokens.matches结果为[nth,child,2,0,2....]  
 // nth-child与nth-of-type有个特殊之处，可以使用p:nth-child(2n),返回的是子元素为p的第2,4,6,8元素  
 // 对于div:nth-child(2)，first为0,last为2；  
 // 对于div:nth-child(2n)，first为2,last为0；  
 "CHILD": function( type, what, argument, first, last ) {  
 var simple = type.slice( 0, 3 ) !== "nth",  
 // 对于$("div:nth-last-child(2)"),type为nth-last  
 forward = type.slice( -4 ) !== "last",  
 // p:nth-of-type(2),选择属于其父元素第二个 <p> 元素的每个 <p> 元素。  
 ofType = what === "of-type";  
  
 return first === 1 && last === 0 ?  
 // 对于$("div:nth-child(1n)")，则为true  
 // 表示选择全部div，故需要有父级元素  
 function( elem ) {  
 return !!elem.parentNode;  
 } :  
  
 function( elem, context, xml ) {  
 var cache, uniqueCache, outerCache, node, nodeIndex, start,  
 dir = simple !== forward ? "nextSibling" : "previousSibling",  
 parent = elem.parentNode,  
 name = ofType && elem.nodeName.toLowerCase(),  
 useCache = !xml && !ofType,  
 diff = false;  
  
 if ( parent ) {  
  
 // :(first|last|only)-(child|of-type)  
 // 非nth模式  
 // 主要是查询elem的nextSibling和previousSibling元素，  
 // 每个元素上都存储了nextSibling和prviousSibling元素，通过判断同辈元素来判断当前元素的位置  
 if ( simple ) {  
 while ( dir ) {  
 node = elem;  
 while ( (node = node[ dir ]) ) {  
 if ( ofType ?  
 node.nodeName.toLowerCase() === name :  
 node.nodeType === 1 ) {  
  
 return false;  
 }  
 }  
 // 如type为only，唯一元素，故需要查询两个方向，只要同辈有元素，则返回false  
 start = dir = type === "only" && !start && "nextSibling";  
 }  
 return true;  
 }  
 // 根据方向判断，开头元素是firstChild还是lastChild  
 start = [ forward ? parent.firstChild : parent.lastChild ];  
  
 // 处理非xml的nth-child,nth-child存储缓存数据在parent元素上  
 if ( forward && useCache ) {  
  
 node = parent;  
 // 如缓存过此node，则直接获取为outerCacher，否则创建一个空对象  
 // 为何要缓存，因为nth针对都是parent相同的child查找  
 outerCache = node[ expando ] || (node[ expando ] = {});  
  
 // Support: IE <9 only  
 // 解决(jQuery gh-1709)bug，  
 uniqueCache = outerCache[ node.uniqueID ] ||  
 (outerCache[ node.uniqueID ] = {});  
  
 cache = uniqueCache[ type ] || [];  
 nodeIndex = cache[ 0 ] === dirruns && cache[ 1 ];  
 diff = nodeIndex && cache[ 2 ];  
 node = nodeIndex && parent.childNodes[ nodeIndex ];  
  
 while ( (node = ++nodeIndex && node && node[ dir ] ||  
  
 // 从开始寻找elem  
 (diff = nodeIndex = 0) || start.pop()) ) {  
  
 // 在node.childNodes中查询elem，查询到后，将次序缓存起来  
 if ( node.nodeType === 1 && ++diff && node === elem ) {  
 uniqueCache[ type ] = [ dirruns, nodeIndex, diff ];  
 break;  
 }  
 }  
  
 } else {  
 // 使用缓存  
 if ( useCache ) {  
 // ...in a gzip-friendly way  
 node = elem;  
 outerCache = node[ expando ] || (node[ expando ] = {});  
  
 // Support: IE <9 only  
 // 解决(jQuery gh-1709)bug，  
 uniqueCache = outerCache[ node.uniqueID ] ||  
 (outerCache[ node.uniqueID ] = {});  
  
 cache = uniqueCache[ type ] || [];  
 nodeIndex = cache[ 0 ] === dirruns && cache[ 1 ];  
 diff = nodeIndex;  
 }  
  
 // xml :nth-child(...)  
 // or :nth-last-child(...) or :nth(-last)?-of-type(...)  
 if ( diff === false ) {  
 // 使用如上类似的循环结构，只是缓存的数据不同  
 while ( (node = ++nodeIndex && node && node[ dir ] ||  
 (diff = nodeIndex = 0) || start.pop()) ) {  
  
 if ( ( ofType ?  
 node.nodeName.toLowerCase() === name :  
 node.nodeType === 1 ) &&  
 ++diff ) {  
  
 // 缓存每个元素的index  
 if ( useCache ) {  
 outerCache = node[ expando ] || (node[ expando ] = {});  
  
 // Support: IE <9 only  
 // 解决(jQuery gh-1709)bug，  
 uniqueCache = outerCache[ node.uniqueID ] ||  
 (outerCache[ node.uniqueID ] = {});  
  
 uniqueCache[ type ] = [ dirruns, diff ];  
 }  
  
 if ( node === elem ) {  
 break;  
 }  
 }  
 }  
 }  
 }  
  
 diff -= last;  
 //  
 return diff === first || ( diff % first === 0 && diff / first >= 0 );  
 }  
 };  
 },  
  
 "PSEUDO": function( pseudo, argument ) {  
 // 伪类不区分大小写  
 // http://www.w3.org/TR/selectors/#pseudo-classes  
 // setFilters继承自pseudos  
 var args,  
 // Expr.setFilters = Expr.filters = Expr.pseudos  
 fn = Expr.pseudos[ pseudo ] || Expr.setFilters[ pseudo.toLowerCase() ] ||  
 Sizzle.error( "unsupported pseudo: " + pseudo );  
  
 // 可以使用createPseudo表明参数需要创建一个过滤器函数，正如Sizzle所做的  
 // 其实是Expr.pseudos是由MarkFunction创建的，返回fn( argument )  
 if ( fn[ expando ] ) {  
 return fn( argument );  
 }  
  
 // 支持旧的签名  
 // ???????????????????????????????????????  
 if ( fn.length > 1 ) {  
 args = [ pseudo, pseudo, "", argument ];  
 return Expr.setFilters.hasOwnProperty( pseudo.toLowerCase() ) ?  
 markFunction(function( seed, matches ) {  
 var idx,  
 matched = fn( seed, argument ),  
 i = matched.length;  
 while ( i-- ) {  
 idx = indexOf( seed, matched[i] );  
 seed[ idx ] = !( matches[ idx ] = matched[i] );  
 }  
 }) :  
 function( elem ) {  
 return fn( elem, 0, args );  
 };  
 }  
  
 return fn;  
 }  
},

### Expr.pseudos----(1959)

* + - 1. innerText与textContent区别：
         1. textContent会获取全部内容，包括<script>和<style>
         2. textContent不会受css样式的影响，不会触发重排
         3. textContent会意识到隐藏元素的文本
         4. innerText，是IE引入的
      2. textContent与innerHtml区别：
         1. textContent有更好的性能，能防止xss攻击

pseudos: {  
 // :not(selector):去除所有与给定选择器匹配的元素  
 // 如input:not(:checked):查询未选中的input  
 "not": markFunction(function( selector ) {  
 // 去掉selector的前后空格，便于创建compile函数  
 var input = [],  
 results = [],  
 matcher = compile( selector.replace( rtrim, "$1" ) );  
  
 return matcher[ expando ] ?  
 markFunction(function( seed, matches, context, xml ) {  
 var elem,  
 unmatched = matcher( seed, null, xml, [] ),  
 i = seed.length;  
  
 // Match elements unmatched by `matcher`  
 while ( i-- ) {  
 if ( (elem = unmatched[i]) ) {  
 seed[i] = !(matches[i] = elem);  
 }  
 }  
 }) :  
 function( elem, context, xml ) {  
 input[0] = elem;  
 // 因为matcher=compile，此处为compile调用  
 matcher( input, null, xml, results );  
 // 不要保留input[0],处理(issue #299)  
 input[0] = null;  
 return !results.pop();  
 };  
 }),  
 // :has(selector)：判断是否有某个选择器  
 // p:has(a),其实就是为selector传入p的context  
 "has": markFunction(function( selector ) {  
 return function( elem ) {  
 return Sizzle( selector, elem ).length > 0;  
 };  
 }),  
 // 匹配包含给定文本的元素  
 "contains": markFunction(function( text ) {  
 text = text.replace( runescape, funescape );  
 return function( elem ) {  
 return ( elem.textContent || elem.innerText || getText( elem ) ).indexOf( text ) > -1;  
 };  
 }),  
 // 选择指定语言的所有元素  
 // div:lang(en)，会匹配<div lang="en"></div>和<p lang="en-us"></p>  
 // 元素是否:lang()选择器表示，仅基于元素的语言值等于标识符C，或者C-  
 // 且C的匹配不区分小写，C不一定是有效的语言name  
 // 总之是根据如下w3.org提供的方式获取具有lang的元素  
 // http://www.w3.org/TR/selectors/#lang-pseudo  
 "lang": markFunction( function( lang ) {  
 // lang值必须是有效的identifier  
 if ( !ridentifier.test(lang || "") ) {  
 Sizzle.error( "unsupported lang: " + lang );  
 }  
 lang = lang.replace( runescape, funescape ).toLowerCase();  
 return function( elem ) {  
 var elemLang;  
 do {  
 if ( (elemLang = documentIsHTML ?  
 elem.lang :  
 elem.getAttribute("xml:lang") || elem.getAttribute("lang")) ) {  
  
 elemLang = elemLang.toLowerCase();  
 return elemLang === lang || elemLang.indexOf( lang + "-" ) === 0;  
 }  
 // ??????????????????????????为何迭代parentNode  
 } while ( (elem = elem.parentNode) && elem.nodeType === 1 );  
 return false;  
 };  
 }),  
 // 选择由文档URI的格式化识别码表示的目标元素  
 // 如输入http://www.xxx.com/#foo,p:target,会查询到<p id=foo></p>  
 "target": function( elem ) {  
 var hash = window.location && window.location.hash;  
 return hash && hash.slice( 1 ) === elem.id;  
 },  
 // 选择该文档的根元素,html文档中选择的是html元素  
 // docElem当前文档  
 "root": function( elem ) {  
 return elem === docElem;  
 },  
  
 "focus": function( elem ) {  
 // activeElement:返回当前页面中获得焦点的元素,如果没有某个元素获得焦点,则该属性的值为当前页面中的<body>元素  
 // document.hasFocus():来判断当前文档中的活动元素是否获得了焦点。  
 // 为何(!document.hasFocus || document.hasFocus())判断：因为opera浏览器未实现此方法  
 // tabIndex:表示元素（如果可聚焦）是否能够接受输入焦点,通过这这tabIndex值，  
 // 可以规定使用键盘方向键或tab键时获取焦点的顺序，可以让div等元素获得焦点  
 // ~tabIndex == tabIndex != -1;~运算，按位取反  
 return elem === document.activeElement && (!document.hasFocus || document.hasFocus()) && !!(elem.type || elem.href || ~elem.tabIndex);  
 },  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*表单对象属性\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 "enabled": createDisabledPseudo( false ),  
 "disabled": createDisabledPseudo( true ),  
 // 匹配所有选中的被选中元素(复选框、单选框等，select中的option)  
 "checked": function( elem ) {  
 // css3中，:checked返回checked和selected元素  
 // http://www.w3.org/TR/2011/REC-css3-selectors-20110929/#checked  
 var nodeName = elem.nodeName.toLowerCase();  
 return (nodeName === "input" && !!elem.checked) || (nodeName === "option" && !!elem.selected);  
 },  
 // 对于select元素来说，获取选中推荐使用 :selected  
  
 "selected": function( elem ) {  
 // 这个主要是为了让Safari默认选项正确  
 if ( elem.parentNode ) {  
 // selectedIndex表示select中选择<option>的索引  
 elem.parentNode.selectedIndex;  
 }  
  
 return elem.selected === true;  
 },  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*内容处理\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 // 匹配所有不包含子元素或者文本的空元素  
 "empty": function( elem ) {  
 // http://www.w3.org/TR/selectors/#empty-pseudo  
 // :empty 对于element (1) or content nodes (text: 3; cdata: 4; entity ref: 5)返回false,  
 // 对于(comment: 8; processing instruction: 7; etc.)返回true  
 // nodeType < 6 可以判断empty，因为attributes (2) 不会出现在子节点  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*提供遍历dom的方法  
 for ( elem = elem.firstChild; elem; elem = elem.nextSibling ) {  
 if ( elem.nodeType < 6 ) {  
 return false;  
 }  
 }  
 return true;  
 },  
 // 匹配含有子元素或者文本的元素  
 "parent": function( elem ) {  
 return !Expr.pseudos["empty"]( elem );  
 },  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*表单元素\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 // 匹配如 h1, h2, h3之类的标题元素  
 "header": function( elem ) {  
 // rheader = /^h\d$/i  
 return rheader.test( elem.nodeName );  
 },  
 // 匹配所有 input, textarea, select 和 button 元素  
 "input": function( elem ) {  
 // rinputs = /^(?:input|select|textarea|button)$/i,  
 return rinputs.test( elem.nodeName );  
 },  
 // 匹配所有按钮  
 // <input type="button" />,<button></button>  
 "button": function( elem ) {  
 var name = elem.nodeName.toLowerCase();  
 return name === "input" && elem.type === "button" || name === "button";  
 },  
 // 查找所有文本框  
 "text": function( elem ) {  
 var attr;  
 return elem.nodeName.toLowerCase() === "input" &&  
 elem.type === "text" &&  
  
 // Support: IE<8  
 // 新的html属性值 (如 "search")会被转换为为elem.type === "text"  
 ( (attr = elem.getAttribute("type")) == null || attr.toLowerCase() === "text" );  
 },  
  
 /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*位置判断\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  
 // 主要是为createPositionalPseudo传递一个函数，选择符合的数组元素  
 "first": createPositionalPseudo(function() {  
 return [ 0 ];//返回数组第一个元素  
 }),  
  
 "last": createPositionalPseudo(function( matchIndexes, length ) {  
 return [ length - 1 ];  
 }),  
  
 "eq": createPositionalPseudo(function( matchIndexes, length, argument ) {  
 return [ argument < 0 ? argument + length : argument ];  
 }),  
  
 "even": createPositionalPseudo(function( matchIndexes, length ) {  
 var i = 0;  
 for ( ; i < length; i += 2 ) {  
 matchIndexes.push( i );  
 }  
 return matchIndexes;  
 }),  
  
 "odd": createPositionalPseudo(function( matchIndexes, length ) {  
 var i = 1;  
 for ( ; i < length; i += 2 ) {  
 matchIndexes.push( i );  
 }  
 return matchIndexes;  
 }),  
 // 匹配所有小于给定索引值的元素  
 "lt": createPositionalPseudo(function( matchIndexes, length, argument ) {  
 var i = argument < 0 ? argument + length : argument;  
 for ( ; --i >= 0; ) {  
 matchIndexes.push( i );  
 }  
 return matchIndexes;  
 }),  
  
 "gt": createPositionalPseudo(function( matchIndexes, length, argument ) {  
 var i = argument < 0 ? argument + length : argument;  
 for ( ; ++i < length; ) {  
 matchIndexes.push( i );  
 }  
 return matchIndexes;  
 })  
}

## DOM筛选与查找方法

### Sizzle.matchesSelector = function( elem, expr )----(L1516)

* + - 1. Sizzle最后将jQuery.find = Sizzle，故sizzle上的方法都会在jQuery.find上面

/\*\*  
 \* 用于检查某个元素node是否匹配选择器表达式expr。  
 \* Sizzle本身未调用，但jQuery通过jqeury.find.matchesSelector进行调用  
 \* @param elem  
 \* @param expr  
 \* @return {\*}  
 \*/  
Sizzle.matchesSelector = function( elem, expr ) {  
 // 如需要设置document  
 if ( ( elem.ownerDocument || elem ) !== document ) {  
 setDocument( elem );  
 }  
  
 // 确保引用了属性选择器  
 expr = expr.replace( rattributeQuotes, "='$1']" );  
 // support.matchesSelector是否存在matches方法,如原生支持，则使用原生方法  
 if ( support.matchesSelector && documentIsHTML &&  
 !compilerCache[ expr + " " ] &&  
 ( !rbuggyMatches || !rbuggyMatches.test( expr ) ) &&  
 ( !rbuggyQSA || !rbuggyQSA.test( expr ) ) ) {  
  
 try {  
 var ret = matches.call( elem, expr );  
  
 // IE 9's matchesSelector在未挂在dom树的nodes返回false  
 if ( ret || support.disconnectedMatch ||  
 // 同样，IE9，未挂在DOM上的节点会被认为在frament上  
 elem.document && elem.document.nodeType !== 11 ) {  
 return ret;  
 }  
 } catch (e) {}  
 }  
  
 return Sizzle( expr, document, null, [ elem ] ).length > 0;  
};

### jQuery.filter = function( expr, elems, not )----(L2887)

* + - 1. $.filter('.rex',$('div'))会返回div.rex的数组，返回的是DOM数组，并不是jquery对象数组，注意DOM数组与jQuery数组



/\*\*  
 \* 使用指定的选择器表达式expr对元素集合elems进行过滤，并返回过滤结果。  
 \* 如果参数not是true，则保留不匹配元素，否则默认保留匹配元素。  
 \* @param expr  
 \* @param elems  
 \* @param not  
 \* @return {\*} 返回DOM数组  
 \*/  
jQuery.filter = function( expr, elems, not ) {  
 var elem = elems[ 0 ];  
  
 if ( not ) {  
 expr = ":not(" + expr + ")";  
 }  
  
 if ( elems.length === 1 && elem.nodeType === 1 ) {  
 // matchesSelector主要是判断浏览器是否之后matches方法，不支持则调用Sizzle函数  
 return jQuery.find.matchesSelector( elem, expr ) ? [ elem ] : [];  
 }  
 //过滤数组，返回新数组,满足后面function的添加到数组中  
 return jQuery.find.matches( expr, jQuery.grep( elems, function( elem ) {  
 return elem.nodeType === 1;  
 } ) );  
};

### find、filter、not、is----(L2903)

* + - 1. function winnow( elements, qualifier, not )----(L2854)

/\*\*  
 \* 提取not与filter函数的公共部分,负责过滤元素集合  
 \* @param elements 待过滤的元素集合  
 \* @param qualifier 用于过滤元素集合elements，可选值有函数、DOM元素、选择器表达式、DOM元素数组、jQuery对象。  
 \* @param not  
 \* @return {\*}  
 \*/  
function winnow( elements, qualifier, not ) {  
 // 如qualifier为函数，则在elements每个元素上调用此函数  
 if ( jQuery.isFunction( qualifier ) ) {  
 return jQuery.grep( elements, function( elem, i ) {  
 // 如not为true，则表示选取elements不满足qualifier的值，故elem不满足时返回false  
 return !!qualifier.call( elem, i, elem ) !== not;  
 } );  
 }  
  
 // 如果参数qualifier是DOM元素，DOM元素，则直接  
 if ( qualifier.nodeType ) {  
 return jQuery.grep( elements, function( elem ) {  
 return ( elem === qualifier ) !== not;  
 } );  
 }  
  
 // elements为Arraylike类型，如 (jQuery, arguments, Array)  
 if ( typeof qualifier !== "string" ) {  
 return jQuery.grep( elements, function( elem ) {  
 return ( indexOf.call( qualifier, elem ) > -1 ) !== not;  
 } );  
 }  
  
 // 简单的selecotr直接调用jQuery.filter  
 if ( risSimple.test( qualifier ) ) {  
 return jQuery.filter( qualifier, elements, not );  
 }  
  
 // 复杂的selector，比较两个数据集  
 // 首先过滤出满足qulifiler的elements集合a，元素elements集合为b  
 // 再在b中查询a  
 qualifier = jQuery.filter( qualifier, elements );  
 return jQuery.grep( elements, function( elem ) {  
 return ( indexOf.call( qualifier, elem ) > -1 ) !== not && elem.nodeType === 1;  
 } );  
}

* + - 1. 源码解析

jQuery.fn.extend( {  
 // 对于$()的某些情况也会调用此方法  
 find: function( selector ) {  
 var i, ret,  
 len = this.length,// 表示$()选择到的dom数量  
 self = this;  
  
 if ( typeof selector !== "string" ) {  
 return this.pushStack( jQuery( selector ).filter( function() {  
 for ( i = 0; i < len; i++ ) {  
 if ( jQuery.contains( self[ i ], this ) ) {  
 return true;  
 }  
 }  
 } ) );  
 }  
  
 ret = this.pushStack( [] );  
  
 for ( i = 0; i < len; i++ ) {  
 // jQuery.find = Sizzle;  
 jQuery.find( selector, self[ i ], ret );  
 }  
 // jQuery.uniqueSort()分类，删除重复  
 return len > 1 ? jQuery.uniqueSort( ret ) : ret;  
 },  
 filter: function( selector ) {  
 return this.pushStack( winnow( this, selector || [], false ) );  
 },  
 not: function( selector ) {  
 return this.pushStack( winnow( this, selector || [], true ) );  
 },  
 // 根据selector来检测匹配元素集合，如果其中至少有一个元素符合这个给定的表达式就返回true  
 // \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*通过判断winnow.length，返回true与false，则判断是否有要检测的对象  
 is: function( selector ) {  
 return !!winnow(  
 this,  
  
 // 如这参数是位置或关系选择器，检查返回集合的元素，保证  
 // $("p:first").is("p:last") 不会返回true  
 // rneedsContext.test( selector )用来匹配位置或关系选择器  
 typeof selector === "string" && rneedsContext.test( selector ) ?  
 jQuery( selector ) :  
 selector || [],  
 false  
 ).length;  
 }  
} );

### has、closest、index、add、addBack-------(L3076)

jQuery.fn.extend( {  
 // 保留包含特定后代的元素  
 // 因为targets为jQuery(target,this)，在this的context下查找target  
 has: function( target ) {  
 // jQuery = function( selector, context )  
 var targets = jQuery( target, this ),  
 l = targets.length;  
  
 return this.filter( function() {  
 var i = 0;  
 for ( ; i < l; i++ ) {  
 if ( jQuery.contains( this, targets[ i ] ) ) {  
 return true;  
 }  
 }  
 } );  
 },  
 // 从元素本身开始，逐级向上级元素匹配，并返回最先匹配的元素  
 closest: function( selectors, context ) {  
 var cur,  
 i = 0,  
 l = this.length,  
 matched = [],  
 targets = typeof selectors !== "string" && jQuery( selectors );  
  
 // 当无selection上下文时，位置选择符不匹配  
 // 即对于jQuery自定义伪类（(even|odd|eq|gt|lt|nth|first|last），直接返回空  
 // 对于parent不存在even等  
 if ( !rneedsContext.test( selectors ) ) {  
 for ( ; i < l; i++ ) {  
 for ( cur = this[ i ]; cur && cur !== context; cur = cur.parentNode ) {  
  
 // 总是跳过framents  
 if ( cur.nodeType < 11 && ( targets ?  
 targets.index( cur ) > -1 :  
  
 // 不要将非elements元素传给Sizzle  
 cur.nodeType === 1 &&  
 jQuery.find.matchesSelector( cur, selectors ) ) ) {  
  
 matched.push( cur );  
 break;  
 }  
 }  
 }  
 }  
  
 return this.pushStack( matched.length > 1 ? jQuery.uniqueSort( matched ) : matched );  
 },  
  
 // 检查集合中某个元素的位置  
 index: function( elem ) {  
  
 // 不传递参数，返回这个元素在同辈中的索引位置。  
 // 对于$()返回集合>1，则$().index()返回的是第一匹配元素位置  
 if ( !elem ) {  
 return ( this[ 0 ] && this[ 0 ].parentNode ) ? this.first().prevAll().length : -1;  
 }  
  
 // 针对elem为选择器  
 // $('.group1').index('p'),返回的是.group1在p中的位置  
 if ( typeof elem === "string" ) {  
 return indexOf.call( jQuery( elem ), this[ 0 ] );  
 }  
  
 // 针对elem为jquery对象  
 // $('p').index($('.group1'))，返回是jquery对象相对于前面p而言的位置  
 return indexOf.call( this,  
  
 // 如果elem为jQuery对象，则使用第一个元素。  
 elem.jquery ? elem[ 0 ] : elem  
 );  
 },  
 // 把与表达式匹配的元素添加到jQuery对象中  
 add: function( selector, context ) {  
 return this.pushStack(  
 jQuery.uniqueSort(  
 // merge是用来合并数组的  
 // 注意：jQuery.merge( this.get(), jQuery( selector, context ) ):返回的是纯数组，因为this.get()获得的是this数组  
 // jQuery.merge( this, jQuery( selector, context ) ):返回的是jQuery对象  
 jQuery.merge( this.get(), jQuery( selector, context ) )  
 )  
 );  
 },  
 // 把堆栈顶层元素加到当前this集合中  
 // $('span').parent().find('a').addBack(),返回的是[div.rex,a]  
 // 如无addBack()则只是a  
 // 传入参数selector，则在preObject中查询此selector  
 addBack: function( selector ) {  
 return this.add( selector == null ?  
 this.prevObject : this.prevObject.filter( selector )  
 );  
 }  
} );

### parent、parents、next、prev、nextAll-------(L3162)

* + - 1. var dir = function( elem, dir, until ) -----------(L2809)

/\*\*  
 \* parent，parents等调用的内部函数  
 \* @param elem  
 \* @param dir 方向，parentNode、nextSibling、previousSibling等  
 \* @param until  
 \* @return {Array}  
 \*/  
var dir = function( elem, dir, until ) {  
 var matched = [],  
 truncate = until !== undefined;  
  
 while ( ( elem = elem[ dir ] ) && elem.nodeType !== 9 ) {  
 if ( elem.nodeType === 1 ) {  
 // 如elem是until则返回，不添加到matched数组中  
 if ( truncate && jQuery( elem ).is( until ) ) {  
 break;  
 }  
 matched.push( elem );  
 }  
 }  
 return matched;  
};

* + - 1. function sibling( cur, dir )------(L3157)

// dir是循环多次，求结果  
// sibling则只是循环找到elements则结束  
function sibling( cur, dir ) {  
 while ( ( cur = cur[ dir ] ) && cur.nodeType !== 1 ) {}  
 return cur;  
}

* + - 1. var siblings = function( n, elem ) ---------------------(L2825)

// 查找同辈元素  
// \*\*\*\*\*\*\*\*\*\*\*\* 并不是在当前元素分别向前和向后查找同辈元素  
// 而是先选取elem.parent.firstChild，然后一直查询nextSibling，并排除自身  
var siblings = function( n, elem ) {  
 var matched = [];  
  
 for ( ; n; n = n.nextSibling ) {  
 if ( n.nodeType === 1 && n !== elem ) {  
 matched.push( n );  
 }  
 }  
  
 return matched;  
};

* + - 1. 源码解析

// dir即是返回elem的（parentNode、previousSibling）数组集合  
jQuery.each( {  
 parent: function( elem ) {  
 var parent = elem.parentNode;  
 return parent && parent.nodeType !== 11 ? parent : null;  
 },  
 parents: function( elem ) {  
 return dir( elem, "parentNode" );  
 },  
 // parentsUntil与parents的区别，不传入参数时，两者一致  
 // 如传入参数，则如parentsUntil('html')不包含html元素，但parents('html')返回数组包含html元素  
 parentsUntil: function( elem, i, until ) {  
 return dir( elem, "parentNode", until );  
 },  
 next: function( elem ) {  
 return sibling( elem, "nextSibling" );  
 },  
 prev: function( elem ) {  
 return sibling( elem, "previousSibling" );  
 },  
 nextAll: function( elem ) {  
 return dir( elem, "nextSibling" );  
 },  
 // 查找当前元素之前所有的同辈元素  
 prevAll: function( elem ) {  
 return dir( elem, "previousSibling" );  
 },  
 nextUntil: function( elem, i, until ) {  
 return dir( elem, "nextSibling", until );  
 },  
 // 找出元素之前的同辈元素  
 prevUntil: function( elem, i, until ) {  
 return dir( elem, "previousSibling", until );  
 },  
 // 找出同辈元素  
 siblings: function( elem ) {  
 return siblings( ( elem.parentNode || {} ).firstChild, elem );  
 },  
 // 所有子元素的元素集合  
 children: function( elem ) {  
 return siblings( elem.firstChild );  
 },  
 // 查找匹配元素内部所有的子节点（包括文本节点）。  
 // 如果元素是一个iframe，则查找文档内容  
 // 因为查找包含文本节点，故不适用siblings  
 contents: function( elem ) {  
 if ( nodeName( elem, "iframe" ) ) {  
 return elem.contentDocument;  
 }  
  
 // Support: IE 9 - 11 only, iOS 7 only, Android Browser <=4.3 only  
 // 在不支持的浏览器中，将模板元素视为常规元素  
 if ( nodeName( elem, "template" ) ) {  
 elem = elem.content || elem;  
 }  
  
 return jQuery.merge( [], elem.childNodes );  
 }  
}, function( name, fn ) {  
 jQuery.fn[ name ] = function( until, selector ) {  
 // 用this通过fn转换为一个matched数组，  
 var matched = jQuery.map( this, fn, until );  
  
 if ( name.slice( -5 ) !== "Until" ) {  
 selector = until;  
 }  
 // 如传入参数，即类似$('span').parents('.rex');故需要将match集合用filter过滤  
 if ( selector && typeof selector === "string" ) {  
 matched = jQuery.filter( selector, matched );  
 }  
  
 if ( this.length > 1 ) {  
  
 // 去重  
 if ( !guaranteedUnique[ name ] ) {  
 jQuery.uniqueSort( matched );  
 }  
  
 // 反序parents与prev开头的  
 if ( rparentsprev.test( name ) ) {  
 matched.reverse();  
 }  
 }  
  
 return this.pushStack( matched );  
 };  
} );

# 回调对象

## 概述

jQuery.Callbacks()是在版本1.7中新加入的，一个多用途的回调列表对象，提供了强大的的方式来管理回调函数列表

### 主要使用场景

控制一系列的函数顺序执行；

最简单的思路是，把函数添加到数组中，当需要时从数组中拿出来进行运行

为了可以增加程序松耦合，jquery使用了基于发布订阅的观察者设计模式，因为使用了观察者模式，故add到list中的函数，当fire时，会将fire的参数全部传入到list中每个函数中，依次获得结果

### 举例

* + - 1. 通常情况下函数队列的处理方式：

// 执行函数  
function runlist(func) {  
 for(var i = 0 ; i <func.length ; i++){  
 func[i]();  
 }  
 func.length = 0;  
}  
// 添加  
var list = [];  
list.push(console1);  
list.push(console2);  
list.push(console3);  
  
runlist(list);  
function console1() {  
 console.log('1')  
}  
function console2() {  
 console.log('2')  
}  
function console3() {  
 console.log('3')  
}

* + - 1. 利用jQuery回调函数

var callback = $.Callbacks();  
callback.add(console1);  
callback.add(console2);  
callback.add(console3);  
callback.fire();

## 源码---（L3274）

### 设置属性分析

* + - 1. Once：确保这个回调列表只执行（.fire() ）一次.

var f1 = function (val) {  
 console.log(val);  
};  
var callback = $.Callbacks('once');  
callback.add(f1);  
callback.fire(1);  
callback.add(f1);  
callback.fire('2');// 结果只会显示1次1

* + - * 1. 如设置once属性，则在fire中会将locked设置为true，因memory为false，故会将list=””，则再add时无法add函数，再调用fire时，fireWith内部也不会调用
      1. Memory:保持以前的值（参数），将函数添加到这个列表的后面，并使用先前保存的参数立即执行该函数。他有一个特点，就是在第一次fire之前使用add添加的回调都不会马上执行，只有调用了一次fire之后使用add添加的回调会马上执行。

var fn1 = function(value) { console.log(value); };  
var callbacks = $.Callbacks("memory");  
callbacks.add( fn1 );//无执行结果  
callbacks.fire( "1" );//执行结果1。保存场景参数1  
console.log('\*\*');  
callbacks.add( fn1 );//执行结果1。使用上次保存的场景参数1  
callbacks.fire( "2" );//执行结果2,2。保存场景参数2

* + - * 1. 第一次调用add，add函数中的memory为undefined，故仅仅是将函数加到list中；当调用fire时，会将此时的参数”1”添加到queue数组中，执行fn1函数
        2. 第二次调用add，会直接fire，利用上一次fire的参数
      1. Unique：确保一次只能添加一个回调(所以在列表中没有重复的回调).通过jQuery.Callbacks里面的has函数，判断list是否已经有func添加
      2. stopOnFalse: 当一个回调返回false 时中断调用

### 源码

/\*  
 \* 一个多用途的回调列表对象，提供了强大的的方式来管理回调函数列表。  
 \* 使用如下参数，构建一个callbasc列表  
 \*  
 \* options: 传入options会改变回调列表的行为方式  
 \*  
 \* 默认的回调列表充当事件回调列表，并且可以fired多次  
 \*  
 \* 可选的options:  
 \*  
 \* once: 确保回调列表只能fired一次（如Deferred）  
 \*  
 \* memory: 保存之前的值，当新函数add到list时，无需fire，自动执行  
 \*  
 \* unique: 确保callback只能被添加一次（list中无重复的）  
 \*  
 \* stopOnFalse: 当回调返回false时，中断调用  
 \*  
 \*/  
jQuery.Callbacks = function( options ) {  
  
 // 如需要将字符串格式的options转为object格式  
 options = typeof options === "string" ?  
 // 将字符串格式的options转为object格式  
 // 类似options=memory，则转换为options.memory = true  
 createOptions( options ) :  
 jQuery.extend( {}, options );  
  
 var // list是否现在firing的标识  
 firing,  
  
 // 对于 non-forgettable lists最后一个fire值  
 // $.Callbacks("memory"),调用过fire后，再add会自动调用fire  
 // add函数中有设置，即memory为true，则调用fire()  
 memory,  
  
 // list是否已经fired  
 fired,  
  
 // 阻止firing的标识  
 locked,  
  
 // 实际的回调列表  
 list = [],  
  
 // 主要是存放memory值,即参数值  
 queue = [],  
  
 // 当前调用的回调index ，如需要会被add或remove修改  
 firingIndex = -1,  
 // fire() 方法作为私有方法被封装在函数中,外部不可直接访问  
 // 因此像 memory、firing、fired 这些状态对于外部上下文来说是不可更改的  
 fire = function() {  
  
 // 如otions.once设置，表示只能fire一次，设置loacked值  
 locked = locked || options.once;  
  
 // 执行list中的全部回调函数  
 fired = firing = true;  
 // 当callback.fire()方法调用时，fireWith会将[context,arugments]，push到queue中，queue.length=1  
 for ( ; queue.length; firingIndex = -1 ) {  
 memory = queue.shift();  
 // 根据fireWith的分析， memory[ 0 ], memory[ 1 ]分别为context与arugments  
 while ( ++firingIndex < list.length ) {  
  
 if ( list[ firingIndex ].apply( memory[ 0 ], memory[ 1 ] ) === false &&  
 options.stopOnFalse ) {  
  
 // stopOnFalse为true，则跳到list最后，  
 // 再调用add，也不会再次fire  
 firingIndex = list.length;  
 memory = false;  
 }  
 }  
 }  
  
 // 如未传入memory参数，则将memory设置为false  
 if ( !options.memory ) {  
 memory = false;  
 }  
  
 firing = false;  
  
 // Clean up if we're done firing for good  
 if ( locked ) {  
  
 // 保留一个空list，为了之后再添加回调时使用  
 if ( memory ) {  
 list = [];  
  
 } else {  
 list = "";  
 }  
 }  
 },  
  
 // 实际的回调对象  
 self = {  
  
 // 添加回调或回调集合到list中  
 add: function() {  
 if ( list ) {  
  
 // 如options为memory参数，应在add后直接firing  
 if ( memory && !firing ) {  
 firingIndex = list.length - 1;  
 queue.push( memory );  
 }  
 // 将add参数arguments传入这个立即运行函数  
 // 如args为([bar,[foo,foo1]])，each循环2次，第一次为bar  
 // 第二次为[foo,foo1]，通过递归，将这个数组再次调用add()  
 // 最后将bar,foo,foo1添加到list中  
 // arguments比较特殊，如参数为[1,2,3],但arguments.length=1,故each不会循环多次  
 ( function add( args ) {  
 jQuery.each( args, function( \_, arg ) {  
 if ( jQuery.isFunction( arg ) ) {  
 // options.unique只能添加一个回调（不能有重复的）  
 if ( !options.unique || !self.has( arg ) ) {  
 list.push( arg );  
 }  
 } else if ( arg && arg.length && jQuery.type( arg ) !== "string" ) {  
 // 如args是嵌套数组，则需要用递归将参数全部加入到list中  
 // 检查递归  
 add( arg );  
 }  
 } );  
 } )( arguments );  
  
 if ( memory && !firing ) {  
 fire();  
 }  
 }  
 return this;  
 },  
  
 // 删除list中的一个回调  
 remove: function() {  
 jQuery.each( arguments, function( \_, arg ) {  
 var index;  
 while ( ( index = jQuery.inArray( arg, list, index ) ) > -1 ) {  
 list.splice( index, 1 );  
  
 // Handle firing indexes  
 if ( index <= firingIndex ) {  
 firingIndex--;  
 }  
 }  
 } );  
 return this;  
 },  
  
 // 检查一个回调是否在list中  
 has: function( fn ) {  
 return fn ?  
 jQuery.inArray( fn, list ) > -1 :  
 list.length > 0;  
 },  
  
 // 清空list中的所有回调  
 // 直接将list=[]则是清空  
 empty: function() {  
 if ( list ) {  
 list = [];  
 }  
 return this;  
 },  
  
 // 禁用 .fire and .add  
 // 与lock主要区别是，不会判断！mermory与！firing  
 disable: function() {  
 locked = queue = [];  
 list = memory = "";  
 return this;  
 },  
 disabled: function() {  
 return !list;  
 },  
  
 // 禁用 .fire  
 // 禁用fire同时会禁用add，除非设置了options=memory  
 // 锁定一个回调列表，以避免进一步的修改列表状态  
 // 即，lock之后，fire不会再有函数运行结果  
 lock: function() {  
 locked = queue = [];  
 if ( !memory && !firing ) {  
 list = memory = "";  
 }  
 return this;  
 },  
 locked: function() {  
 return !!locked;  
 },  
  
 // 访问给定的上下文和参数列表中的所有回调  
 fireWith: function( context, args ) {  
 // callback调用lock()方法后，locked=[]  
 if ( !locked ) {  
 args = args || [];  
 args = [ context, args.slice ? args.slice() : args ];  
 queue.push( args );  
 if ( !firing ) {  
 fire();  
 }  
 }  
 return this;  
 },  
  
 // 用给定的arguments调用list中的所有回调函数  
 fire: function() {  
 self.fireWith( this, arguments );  
 return this;  
 },  
  
 // 判断list是否调用过fire  
 fired: function() {  
 return !!fired;  
 }  
 };  
  
 return self;  
};

# Deferred对象(异步队列)

## 概述

### Deferreds对象

jQuery1.5中新增的Deferreds对象，可以将任务完成的处理方式与任务本身解耦合。举一个简单例子，因为异步导致流程并不是输出1,2,3，Deferreds 对这个问题提供了一个更好的解决方案，它是非阻塞的，并且与代码完全解耦 。

Promise/A只是一种规范（通过一组API规范异步化操作，让异步操作流程控制更加容易），Deferred可以看作这种规范的具体实现;

**相当于异步统一管理**

主要作用：1、解决时序以及动态添加执行函数的问题；

alert(1)  
setTimeout(function(){alert(2)},0)  
alert(3)  
//alert(1)   
//alert(3)  
//alert(2)

2、解决参数传递的问题，比如所有函数需要相同的参数

* + - 1. 每次调用runList时，都需要传递list与val两个参数，这种重复劳动没有任何意义。

var list = [],  
 val = 0;  
list[list.length] = a;  
list[list.length] = b;  
function runList(listArray,value){  
 for(var i = 0; i < listArray.length; i++){  
 listArray[i](value);  
 }  
 listArray.length = 0;  
}  
val = 5;  
runList(list,val );//执行a = 5;b = 5;

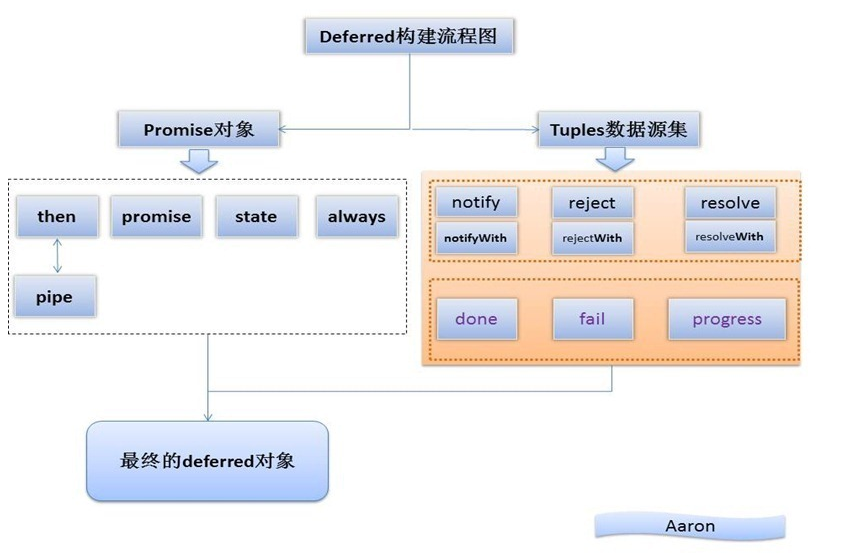
list[list.length] = function c(value){alert("c = " + value)};  
runList(list,val);//执行结果c = 5

### Promises

大多情况下，promise作为一个模型，提供了一个在软件工程中描述延时（或将来）概念的解决方案。

背后的思想是：不是执行一个方法然后阻塞应用程序等待结果返回，而是返回一个promise对象来满足未来值。

### 构建Deffered对象流程图



### jQuery涉及到promise的相关实现都不很理解

## Deferred对象方法简介

### $.Deferred() :生成一个deferred对象。

### Done、fail、then:

成功时回调与失败的回调、准确说应该是添加成功或失败回调；

then将成功与失败回调整合在一起

### Resolve():手动改变deferred对象运行状态为“已完成”，从而触发done()方法

### Reject():手动改变deferred对象运行状态为“已失效”，从而触发fail()方法

### Always()：无论调用resolve()还是reject()，总是执行

## Deferred对象的基本用法

* + - 1. 输出顺序，aaa,111,bbb

function runAsync(){  
 var def = $.Deferred();  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 def.resolve('bbb');  
 }, 0);  
 return def;  
}  
runAsync().then(function(data){  
 console.log('1111');  
 console.log(data)  
});

* + - 1. ES6的promise实现方式

function runAsync(){  
 var p = new Promise(function(resolve, reject){  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 resolve('bbb');  
 }, 0);  
 });  
 return p;  
}  
runAsync().then(function (data) {  
 console.log('1111');  
 console.log(data)  
});

* + - * 1. 与jQuery的Deferred重要区别，deferred上具有resolve方法，可以在外部改变def状态
      1. 外部修改deferred状态
         1. 输出结果：aaaa,111, 在外部结束

function runAsync(){  
 var def = $.Deferred();  
 //做一些异步操作  
 setTimeout(function(){  
 console.log('aaa');  
 def.resolve('bbb');  
 }, 0);  
 return def;  
}  
var d = runAsync();  
d.then(function(data){  
 console.log('1111');  
 console.log(data)  
});  
d.resolve('在外部结束');

* + - 1. 通过def.promise()避免这样的问题出现
         1. 在(1)中的runAsync中，返回def.promise,这样就不能从外部修改状态，调用d.resolve会报错

## 源码分析

### 总览

Callback被剥离之后，deffered非常精简，精简之后为

jQuery.extend( {  
 Deferred: function( func ){

var tuples = [];

promise = {  
 state: function(){};

always: function(){};

………………

}

// 合并promise到deferred

promise.promise( deferred );

}

when: function( singleValue ){}

} );

* + - * 1. Deferred是个工厂类，返回的是内部构建的deferred对象；
        2. tuples 创建三个$.Callbacks对象，分别表示成功，失败，处理中三种状态；
        3. 创建了一个promise对象，具有state、always、then、primise方法
        4. 扩展primise对象生成最终的Deferred对象，返回该对象

### 源码（L3469）

* + - 1. 实现并不理解，仅了解大概，认为是promiseA规范下的回调队列

function Identity( v ) {  
 return v;  
}  
function Thrower( ex ) {  
 throw ex;  
}  
// jQuery.when使用的回调工厂  
function adoptValue( value, resolve, reject, noValue ) {  
 var method;  
  
 try {  
  
 // 检查value值是否为promise，Check for promise aspect first to privilege synchronous behavior  
 if ( value && jQuery.isFunction( ( method = value.promise ) ) ) {  
 method.call( value ).done( resolve ).fail( reject );  
  
 // 其他 thenables值类型  
 } else if ( value && jQuery.isFunction( ( method = value.then ) ) ) {  
 method.call( value, resolve, reject );  
  
 // 非thenables值类型  
 } else {  
  
 // 通过数组控制 `resolve`参数个数  
 // slice函数会将boolean的noValue值转换为integer  
 // \* false: [ value ].slice( 0 ) => resolve( value )  
 // \* true: [ value ].slice( 1 ) => resolve()  
 resolve.apply( undefined, [ value ].slice( noValue ) );  
 }  
  
 // 对于Promises/A+, 将exceptions转换为rejections  
 // 因为jQuery.when并不会展开thenables，我们可以跳过额外的检查，然后有条件的拒绝  
 } catch ( value ) {  
  
 // Support: Android 4.0 only  
 // 严格模式下，函数调用.call/.apply获取不到全局对象上下文  
 reject.apply( undefined, [ value ] );  
 }  
}  
  
jQuery.extend( {  
  
 Deferred: function( func ) {  
 // 创建三个$.Callbacks对象，分别表示成功，失败，处理中三种状态  
 // 创建过程在下面的jQuery.each，此数组只是抽取公共部分，方便处理  
 var tuples = [  
  
 // action, add listener, callbacks,  
 // ... .then handlers, argument index, [final state]  
 // "memomry"：保存之前的值，当新函数add到list时，无需fire，自动执行  
 // "once" : 确保回调列表只能fired一次（如Deferred）  
 [ "notify", "progress", jQuery.Callbacks( "memory" ),  
 jQuery.Callbacks( "memory" ), 2 ],  
 [ "resolve", "done", jQuery.Callbacks( "once memory" ),  
 jQuery.Callbacks( "once memory" ), 0, "resolved" ],  
 [ "reject", "fail", jQuery.Callbacks( "once memory" ),  
 jQuery.Callbacks( "once memory" ), 1, "rejected" ]  
 ],  
 state = "pending", // deferred默认状态  
 // promise是deferred的一种简化形式（去掉了改变状态的接口）  
 promise = {  
 state: function() {  
 return state;  
 },  
 // 函数列表总是执行  
 always: function() {  
 deferred.done( arguments ).fail( arguments );  
 return this;  
 },  
 "catch": function( fn ) {  
 return promise.then( null, fn );  
 },  
  
 // 保留 pipe 函数以备用  
 pipe: function( /\* fnDone, fnFail, fnProgress \*/ ) {  
 var fns = arguments;  
 // newDefer可以看作是newDefer = $.Deferred();  
 return jQuery.Deferred( function( newDefer ) {  
 jQuery.each( tuples, function( i, tuple ) {  
  
 // Map tuples (progress, done, fail) to arguments (done, fail, progress)  
 var fn = jQuery.isFunction( fns[ tuple[ 4 ] ] ) && fns[ tuple[ 4 ] ];  
 // 给父deferred对象的[ done | fail | progress ]方法都增加一个过滤函数的方法  
 // deferred.progress(function() { bind to newDefer or newDefer.notify })  
 // deferred.done(function() { bind to newDefer or newDefer.resolve })  
 // deferred.fail(function() { bind to newDefer or newDefer.reject })  
 deferred[ tuple[ 1 ] ]( function() {  
 var returned = fn && fn.apply( this, arguments );  
 if ( returned && jQuery.isFunction( returned.promise ) ) {  
 returned.promise()  
 .progress( newDefer.notify )  
 .done( newDefer.resolve )  
 .fail( newDefer.reject );  
 } else {  
 newDefer[ tuple[ 0 ] + "With" ](  
 this,  
 fn ? [ returned ] : arguments  
 );  
 }  
 } );  
 } );  
 fns = null;  
 } ).promise();  
 },  
 /\*\*  
 \* ?????????????????????对实现并不理解，关键是其中resolve函数是实现promiseA规范的  
 \* 理解的则是deferred是具有promise特性的callback回调序列  
 \* @param onFulfilled resolve时调用的回调函数  
 \* @param onRejected reject时调用的回调函数  
 \* @param onProgress progress()方法调用的回调函数。  
 \* @return {\*} deferred.promise()即deferred的简易版本  
 \*/  
 then: function( onFulfilled, onRejected, onProgress ) {  
 var maxDepth = 0;  
 // 在then的return中调用了resolve函数，以第一个举例  
 // depth:0,deferred=newDefer，handler：onProgress,special:newDefer.notifyWith  
 function resolve( depth, deferred, handler, special ) {  
 return function() {  
 var that = this,  
 args = arguments,  
 mightThrow = function() {  
 var returned, then;  
  
 // Support: Promises/A+ section 2.3.3.3.3  
 // https://promisesaplus.com/#point-59  
 // 忽略多次解决尝试  
 if ( depth < maxDepth ) {  
 return;  
 }  
  
 returned = handler.apply( that, args );  
  
 // Support: Promises/A+ section 2.3.1  
 // https://promisesaplus.com/#point-48  
 // If promise and x refer to the same object, reject promise with a TypeError as the reason.  
 if ( returned === deferred.promise() ) {  
 throw new TypeError( "Thenable self-resolution" );  
 }  
  
 // Support: Promises/A+ sections 2.3.3.1, 3.5  
 // https://promisesaplus.com/#point-54 Let then be x.then  
 // https://promisesaplus.com/#point-75  
 // 仅检索 `then` 一次  
 then = returned &&  
  
 // Support: Promises/A+ section 2.3.4  
 // https://promisesaplus.com/#point-64  
 // If x is not an object or function, fulfill promise with x.  
 ( typeof returned === "object" ||  
 typeof returned === "function" ) &&  
 returned.then;  
  
 // Handle a returned thenable  
 if ( jQuery.isFunction( then ) ) {  
  
 // Special processors (notify) just wait for resolution  
 if ( special ) {  
 then.call(  
 returned,  
 resolve( maxDepth, deferred, Identity, special ),  
 resolve( maxDepth, deferred, Thrower, special )  
 );  
  
 // Normal processors (resolve) also hook into progress  
 } else {  
  
 // ...and disregard older resolution values  
 maxDepth++;  
  
 then.call(  
 returned,  
 resolve( maxDepth, deferred, Identity, special ),  
 resolve( maxDepth, deferred, Thrower, special ),  
 resolve( maxDepth, deferred, Identity,  
 deferred.notifyWith )  
 );  
 }  
  
 // Handle all other returned values  
 } else {  
  
 // Only substitute handlers pass on context  
 // and multiple values (non-spec behavior)  
 if ( handler !== Identity ) {  
 that = undefined;  
 args = [ returned ];  
 }  
  
 // Process the value(s)  
 // Default process is resolve  
 ( special || deferred.resolveWith )( that, args );  
 }  
 },  
  
 // Only normal processors (resolve) catch and reject exceptions  
 process = special ?  
 mightThrow :  
 function() {  
 try {  
 mightThrow();  
 } catch ( e ) {  
  
 if ( jQuery.Deferred.exceptionHook ) {  
 jQuery.Deferred.exceptionHook( e,  
 process.stackTrace );  
 }  
  
 // Support: Promises/A+ section 2.3.3.3.4.1  
 // https://promisesaplus.com/#point-61  
 // Ignore post-resolution exceptions  
 if ( depth + 1 >= maxDepth ) {  
  
 // Only substitute handlers pass on context  
 // and multiple values (non-spec behavior)  
 if ( handler !== Thrower ) {  
 that = undefined;  
 args = [ e ];  
 }  
  
 deferred.rejectWith( that, args );  
 }  
 }  
 };  
  
 // Support: Promises/A+ section 2.3.3.3.1  
 // https://promisesaplus.com/#point-57  
 // 立即重新承诺，避免错误的拒绝。  
 if ( depth ) {  
 process();  
 } else {  
  
 // Call an optional hook to record the stack, in case of exception  
 // since it's otherwise lost when execution goes async  
 if ( jQuery.Deferred.getStackHook ) {  
 process.stackTrace = jQuery.Deferred.getStackHook();  
 }  
 window.setTimeout( process );  
 }  
 };  
 }  
 // 最终返回的是$.Deferred(func).promise();  
 // 根据deferred对象构建中，func.call(deferred,deferred);  
 // 因此newDefer指向构建好的jQuery.Deferred对象  
 return jQuery.Deferred( function( newDefer ) {  
  
 // progress\_handlers.add( ... )  
 tuples[ 0 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onProgress ) ?  
 onProgress :  
 Identity,// 返回参数  
 newDefer.notifyWith  
 )  
 );  
  
 // fulfilled\_handlers.add( ... )  
 tuples[ 1 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onFulfilled ) ?  
 onFulfilled :  
 Identity  
 )  
 );  
  
 // rejected\_handlers.add( ... )  
 tuples[ 2 ][ 3 ].add(  
 resolve(  
 0,  
 newDefer,  
 jQuery.isFunction( onRejected ) ?  
 onRejected :  
 Thrower  
 )  
 );  
 } ).promise();// 返回新的promise对象  
 },  
  
 // 为deferred对象获得一个promise  
 // 如obj设置了，则将deferred对象添加到这个对象上  
 // 未传入参数则直接返回promise  
 promise: function( obj ) {  
 return obj != null ? jQuery.extend( obj, promise ) : promise;  
 }  
 },  
 deferred = {};  
  
 // 针对tuples 元素集  
 // 把相同有共同特性的代码的给合并成一种结构，然后通过一次处理  
 // 分别创建a.done,a.progress,a.fail(var a = $.Deferred();)  
 jQuery.each( tuples, function( i, tuple ) {  
 // i= 0时，为[ "notify", "progress", jQuery.Callbacks( "memory" ),jQuery.Callbacks( "memory" ),2 ]  
 // list = tuple[ 2 ]，相当于$.Callbacks("once memory");这一列  
 // callbacks数组,创建不同的Callback对象  
 // 假设三个列表分别为,progressCallback（进度回调列表）,doneCallback（成功回调列表）,failCallback（失败回调列表）  
 var list = tuple[ 2 ],  
 stateString = tuple[ 5 ];// 最终状态，只有i=1,i=2存在  
  
 // promise.progress = progressCallback.add  
 // promise.done = doneCallback.add  
 // promise.fail = failCallback.add  
 // 就是为promise赋jQuery.Callbacks  
 promise[ tuple[ 1 ] ] = list.add;// $.Callback()回调对象的add方法  
  
 // 处理状态，为resolved和rejected两种状态的列表添加预设回调函数  
 if ( stateString ) {  
 // 将3个函数添加到list中  
 list.add(  
 function() {// 为list添加当前状态标识函数  
  
 // state = "resolved" (i.e., fulfilled)  
 // state = "rejected"  
 state = stateString;  
 },  
  
 // 为doneCallback，添加failCallback.disable  
 // 为failCallback，添加为doneCallback.disable  
 // 主要目的是禁止修改状态，如已经成功(i=1时)，则需要将reject列表禁用，不能再执行失败函数  
 tuples[ 3 - i ][ 2 ].disable,// 禁用 .fire and .add  
  
 // progress\_callbacks.lock  
 // 锁定notify列表，即notify列表fire方法不能被调用，但因为callback的options为memory  
 // 故还可以调用add方法  
 tuples[ 0 ][ 2 ].lock // 禁用 .fire  
 );  
 }  
  
 // progress\_handlers.fire  
 // fulfilled\_handlers.fire  
 // rejected\_handlers.fire  
 // 这之后，相当于为list添加了4个函数（对于有stateString状态的）  
 list.add( tuple[ 3 ].fire );  
  
 // deferred.notify = function() { deferred.notifyWith(...) }  
 // deferred.resolve = function() { deferred.resolveWith(...) }  
 // deferred.reject = function() { deferred.rejectWith(...) }  
 // 执行deferred.notifyWith(...)并返回this  
 deferred[ tuple[ 0 ] ] = function() {  
 deferred[ tuple[ 0 ] + "With" ]( this === deferred ? undefined : this, arguments );  
 return this;  
 };  
  
 // deferred.notifyWith = list.fireWith  
 // deferred.resolveWith = list.fireWith  
 // deferred.rejectWith = list.fireWith  
 // fireWith:访问给定的上下文和参数列表中的所有回调  
 // resolveWith/rejectWith/notifyWith 是 callbacks.fireWith 队列方法引用  
 deferred[ tuple[ 0 ] + "With" ] = list.fireWith;  
 } );  
 // 上面foreach整个循环完，deferred包含notify,notifyWith,reject,rejectWith,resolve,resolveWith,6个函数  
 // promise对象增加了progress，done，fail三个函数(其实就是list.add函数)，done,fail因为是Callback，里面list有4个函数  
  
 // 将3个tuples全部处理完后的promise合并到deferred  
 promise.promise( deferred );  
  
 // 如传入参数则调用，then方法就传入了func  
 // 则将构建好的deferred作为执行对象和参数，在then方法中就有调用  
 if ( func ) {  
 func.call( deferred, deferred );  
 }  
  
 // 返回deferred对象  
 return deferred;  
 //小结，  
 // 因此deferred.done,deferred.fail,deferred.progress其实就是不同$.callbacks().add()方法  
 // deferred.resolve,deferred.reject,deferred.notify，其实是调用的$.callback().fireWith()  
 // promise对象是deferred的简易版，没有上述6个函数，故无法改变状态  
 },  
  
 /\*\*  
 \* 提供一种方法来执行一个或多个对象的回调函数，返回这些对象的延时（Deferred）对象。  
 \* 1、无参数， jQuery.when()将返回一个resolved（解决）状态的promise对象。  
 \*  
 \*/  
 when: function( singleValue ) {  
 var  
  
 // 参数个数  
 remaining = arguments.length,  
  
 // 未处理的参数个数  
 i = remaining,  
  
 resolveContexts = Array( i ),  
 resolveValues = slice.call( arguments ),// 将参数类数组转换为数组  
  
 // 主Deferred对象  
 master = jQuery.Deferred(),  
  
 // 次要回调工厂  
 updateFunc = function( i ) {  
 return function( value ) {  
 resolveContexts[ i ] = this;  
 resolveValues[ i ] = arguments.length > 1 ? slice.call( arguments ) : value;  
 if ( !( --remaining ) ) {  
 master.resolveWith( resolveContexts, resolveValues );//相当于fire  
 }  
 };  
 };  
  
 // when无参数或单个参数，采用Promise.resolve  
 // 如无参数，则等价于：master.done( updateFunc( i ) ).resolve()  
 if ( remaining <= 1 ) { // i = remaining  
 adoptValue( singleValue, master.done( updateFunc( i ) ).resolve, master.reject,  
 !remaining );//如无参数则！remaining为true  
  
 // 使用 .then() to 展开thenables (cf. gh-3000)  
 if ( master.state() === "pending" ||  
 jQuery.isFunction( resolveValues[ i ] && resolveValues[ i ].then ) ) {  
  
 return master.then();  
 }  
 }  
  
 // 多参数形式，就如同Promise.all  
 while ( i-- ) {  
 adoptValue( resolveValues[ i ], updateFunc( i ), master.reject );  
 }  
  
 return master.promise();  
 }  
} );

## jQuery.ready实现

* + - 1. jQuery.ready是基于Deferred实现的

// deferred用于DOM ready  
var readyList = jQuery.Deferred();  
  
jQuery.fn.ready = function( fn ) {  
  
 readyList  
 .then( fn ) // 如果fn出错则用jQuery默认的ready异常处理方式  
  
 // 将 jQuery.readyException 包在函数内；为了错误处理时查找错误，而不是用回调方式  
 .catch( function( error ) {  
 jQuery.readyException( error );  
 } );  
  
 return this;  
};  
  
jQuery.extend( {  
  
 // 是否DOM可以被使用，仅设置true一次  
 isReady: false,  
  
 // 计数器，用于计量ready事件发生前需要等待多少项  
 // #6781  
 readyWait: 1,  
  
 ready: function( wait ) {  
  
 // 如wait  
 if ( wait === true ? --jQuery.readyWait : jQuery.isReady ) {  
 return;  
 }  
  
 // 设置DOM已经准备好的标志 　  
 jQuery.isReady = true;  
  
 // 一直等到readyWait为0  
 if ( wait !== true && --jQuery.readyWait > 0 ) {  
 return;  
 }  
  
 // 执行绑定的延时事件 　　  
 readyList.resolveWith( document, [ jQuery ] );  
 }  
} );  
  
jQuery.ready.then = readyList.then;  
  
// completed方法，删除事件监听，调用ready方法  
function completed() {  
 document.removeEventListener( "DOMContentLoaded", completed );  
 window.removeEventListener( "load", completed );  
 jQuery.ready();  
}  
  
// 处理$(document).ready()在浏览器事件触发之后调用的情况  
// Support: IE <=9 - 10 only  
// 某些老的ie浏览器可能会出现  
if ( document.readyState === "complete" ||  
 ( document.readyState !== "loading" && !document.documentElement.doScroll ) ) {  
  
 // 异步处理，使脚本有机会延迟  
 window.setTimeout( jQuery.ready );  
  
} else {  
  
 // 标准浏览器DOM加载完后会触发“DOMContentLoaded”事件  
 document.addEventListener( "DOMContentLoaded", completed );  
  
 // 如没有DOMContentLoaded事件，则回退监听load事件，这个事件会在图片等都加载完响应  
 window.addEventListener( "load", completed );  
}

# 数据缓存

## 概述

### 内存泄露

* + - 1. 循环引用很常见且大部分情况下是无害的，但当参与循环引用的对象中有DOM对象或者ActiveX对象时，循环引用将导致内存泄露。如下为普通的循环引用，引用自身，大部分情况无害；但将new Object替换成document.getElementById或者document.createElement就会发生内存泄露了。

var a=new Object;  
a.r=a;

### 数据缓存

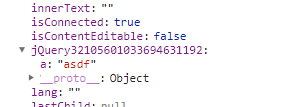
* + - 1. 一般jQuery开发，我们都喜欢便捷式的把很多属性，比如状态标志都写到dom节点中,也就是HTMLElement，这样做好处是直观，坏处则是：
         1. 循环引用
         2. 直接暴露数据，安全性？
         3. 增加一堆的自定义属性标签，对浏览器来说是没意义的
         4. 取数据的时候要对HTML节点做操作
      2. **更好的解决方法是使用一种低耦合的方式让DOM和缓存数据能够联系起来。**
      3. jQuery引入缓存的作用
         1. 允许我们在DOM元素上附加任意类型的数据,避免了循环引用的内存泄漏风险
         2. 用于存储跟dom节点相关的数据，包括事件，动画等
         3. 一种低耦合的方式让DOM和缓存数据能够联系起来

### jQuery实现缓存思路

* + - 1. 在对象上绑定expando属性，并jQuery定义了一个属性cache = {}来保存所有的缓存数据
      2. 通过expando来查找jQuery.cache上对应的缓存数据
      3. 为了保证了id 的全局唯一性，这个id使用jQuery.guid自增

## jQuery.data()与.data()区别

* + - 1. 看jQuery.data(element,[key],[value])，每一个element都会有自己的一个{key:value}对象保存着数据，所以新建的对象就算有key相同它也不会覆盖原来存在的对象key所对应的value，因为新对象保存是是在另一个{key:value}对象中；并且每次调用$.data()，内部data函数，会创建this.expando是不同的
      2. $("div").data("a","aaaa") 它是把数据绑定每一个匹配div节点的元素上，因此相同的key会相互覆盖
      3. 数据存储的位置：



## 源码分析

/\*\*  
 \* jQuery很多函数具有写入和读取功能，如jQuery.fn.css等，  
 \* 此函数为抽象出的入口函数；有时jQuery可以传入obj，或者传入function等，很多都是通过这个函数进行处理的  
 \* 以jquery.fn.extend({data:..})返回access为例  
 \* access首先重要逻辑是处理key为obj形式，如key为obj，则将key再次调用access  
 \* 之后，将处理逻辑分为有key还是无key；如有key，则bulk为false  
 \* 注：用此方法包装的函数调用，提供了四种传值方式：name，(name,value)，({key:value,key:value})，(name,function(index,attr){})

\* @param elems jquery匹配的dom元素  
 \* @param fn fn函数  
 \* @param key null  
 \* @param value value  
 \* @param chainable arguments.length > 1  
 \* @param emptyGet null  
 \* @param raw true  
 \* @return {\*}  
 \*/  
  
var access = function( elems, fn, key, value, chainable, emptyGet, raw ) {  
 var i = 0,  
 len = elems.length,  
 // 通过bulk将处理逻辑分为2大类，key有值，bulk为false，否则为true  
 // 这样封装，主要是为了两类能使用相同的逻辑  
 bulk = key == null; // == 优先级高于=，如key==null，则将bulk赋值为true  
  
 // 如key为object，则递归方法，一个个key设置  
 // 类似，$('a').css({'height':'10px','width':'10px'})  
 // 具体为何不用仔细理解，jQuery.fn.css代码分析会有，jQuery.fn.css内部调用了access,  
 if ( jQuery.type( key ) === "object" ) {  
 chainable = true;  
 for ( i in key ) {  
 access( elems, fn, i, key[ i ], true, emptyGet, raw );  
 }  
  
 // 如value有值，即类似$('a').css({'height':'10px'})的调用方式，key为height，value为10px  
 } else if ( value !== undefined ) {  
 chainable = true;  
 // 如value不是函数，raw设为true，raw标识为value是否为函数  
 if ( !jQuery.isFunction( value ) ) {  
 raw = true;  
 }  
 // key==null，则将bulk赋值为true  
 if ( bulk ) {  
  
 // value不是函数  
 // Bulk 操作针对整个集合operations run against the entire set  
 if ( raw ) {  
 fn.call( elems, value );  
 fn = null;  
  
 // value为函数，elem为普通值，用jquery封装后重新定义fn  
 } else {  
 bulk = fn;  
 fn = function( elem, key, value ) {  
 return bulk.call( jQuery( elem ), value );  
 };  
 }  
 }  
  
 if ( fn ) {  
 for ( ; i < len; i++ ) {  
 fn(  
 elems[ i ], key, raw ?  
 value :  
 value.call( elems[ i ], i, fn( elems[ i ], key ) )  
 );  
 }  
 }  
 }  
 // 如value !== undefined，即有value值都是链式调用；  
 // 上面会将chainable设为true， 则返回elems  
 if ( chainable ) {  
 return elems;  
 }  
  
 // key有值，bulk为false  
 if ( bulk ) {  
 return fn.call( elems ); //以jquery.fn.extend({data:..})返回access为例,则fun的参数value为elems  
 }  
 // 如elem.length不存在，则返回emptyGet  
 return len ? fn( elems[ 0 ], key ) : emptyGet;  
};

/\*\*  
 \* 判断绑定数据的目标owner类型是否符合  
 \* @param owner  
 \* @return {boolean}  
 \*/  
var acceptData = function( owner ) {  
  
 // Accepts only:  
 // - Node  
 // - Node.ELEMENT\_NODE  
 // - Node.DOCUMENT\_NODE  
 // - Object  
 // - Any +owner.nodeType转换为number，但+{}.nodeType === NaN !NaN = true  
 return owner.nodeType === 1 || owner.nodeType === 9 || !( +owner.nodeType );  
};  
  
  
  
  
function Data() {  
 this.expando = jQuery.expando + Data.uid++;  
}  
  
Data.uid = 1;  
/\*\*  
 \* 数据缓存，先在jQuery内部创建一个cache对象{}, 来保存缓存数据。   
 \* 然后往需要进行缓存的DOM节点上扩展一个值为expando的属性  
 \* @type {{  
 \* cache: Data.cache, 创建cache缓存  
 \* set: Data.set, 在owner设置data  
 \* get: Data.get, 获取数据  
 \* access: Data.access, 提供set，get统一访问接口，根据不同情况，调用get，set方法  
 \* remove: Data.remove, 删除值，删除key对应的值  
 \* hasData: Data.hasData 判断owner是否包含数据  
 \* }}  
 \*/  
Data.prototype = {  
 // 创建cache缓存  
 cache: function( owner ) {  
  
 // 检查DOM对象是否有expando属性，判断对象是否有cache  
 var value = owner[ this.expando ];  
  
 // 如无cache，创建一个  
 if ( !value ) {  
 value = {};  
  
 // 现代浏览器可以接收非元素节点绑定数据，但由于see #8335.问题，返回空对象  
 if ( acceptData( owner ) ) {  
  
 // 如owner是元素节点，就在expando上绑定value值  
 if ( owner.nodeType ) {  
 owner[ this.expando ] = value;  
  
 // 将owner的this.expando配置为非枚举属性，为了保证当数据删除时需要将属性同时删除，  
 } else {  
 Object.defineProperty( owner, this.expando, {  
 value: value,  
 configurable: true  
 } );  
 }  
 }  
 }  
  
 return value;  
 },  
 // 在owner设置data，参数可以为[owner, data, value],或者[owner,{properties}]  
 set: function( owner, data, value ) {  
 var prop,  
 cache = this.cache( owner );  
  
 // Handle: [ owner, key, value ] args  
 // 总是使用驼峰标记法的key (gh-2257)  
 if ( typeof data === "string" ) {  
 cache[ jQuery.camelCase( data ) ] = value;  
  
 // Handle: [ owner, { properties } ] args  
 } else {  
  
 // 复制属性到cache对象  
 for ( prop in data ) {  
 cache[ jQuery.camelCase( prop ) ] = data[ prop ];  
 }  
 }  
 return cache;  
 },  
 // 获取数据，如key==undefined，直接返回全部数据，  
 get: function( owner, key ) {  
 return key === undefined ?  
 this.cache( owner ) :  
  
 // 总是使用驼峰标记法 (gh-2257)  
 owner[ this.expando ] && owner[ this.expando ][ jQuery.camelCase( key ) ];  
 },  
 // 提供set，get统一访问接口，根据不同情况，调用get，set方法  
 access: function( owner, key, value ) {  
  
 // 处理如下情况：  
 // 1、未指定key  
 // 2、字符串类型key指定，但未提供value值  
 //  
 // 利用get方法决定何值返回  
 if ( key === undefined ||  
 ( ( key && typeof key === "string" ) && value === undefined ) ) {  
  
 return this.get( owner, key );  
 }  
  
 // 如key不是String，或key与value都指定了，利用set方法，设置值  
 this.set( owner, key, value );  
 // 根据不同情况返回不同值  
 return value !== undefined ? value : key;  
 },  
 // 删除值，删除key对应的值，当key=undefined时，删除owner[ this.expando ]  
 remove: function( owner, key ) {  
 var i,  
 cache = owner[ this.expando ];  
  
 if ( cache === undefined ) {  
 return;  
 }  
  
 if ( key !== undefined ) {  
  
 // 支持keys数组，与空格分隔字符串  
 if ( Array.isArray( key ) ) {  
  
 // 如key是数组，将key全部转换为驼峰标记法的  
 key = key.map( jQuery.camelCase );  
 } else {  
 key = jQuery.camelCase( key );  
  
 // 如key具有空格，则直接使用它  
 // 否则构造一个无空格的array  
 key = key in cache ?  
 [ key ] :  
 ( key.match( rnothtmlwhite ) || [] );  
 }  
  
 i = key.length;  
  
 while ( i-- ) {  
 delete cache[ key[ i ] ];  
 }  
 }  
  
 // 如未传入key或cache无数据，则删除expando  
 //??????????????????????????????????当key为undefined，只是删除了dom的expando属性，未在cache中删除对应数据？？？？？？？  
 if ( key === undefined || jQuery.isEmptyObject( cache ) ) {  
  
 // Support: Chrome <=35 - 45  
 // Webkit & Blink 当DOM节点删除属性时会造成性能下降，故将属性设置为undefined  
 // https://bugs.chromium.org/p/chromium/issues/detail?id=378607 (bug restricted)  
 if ( owner.nodeType ) {  
 owner[ this.expando ] = undefined;  
 } else {  
 delete owner[ this.expando ];  
 }  
 }  
 },  
 // 判断owner是否包含数据  
 hasData: function( owner ) {  
 var cache = owner[ this.expando ];  
 return cache !== undefined && !jQuery.isEmptyObject( cache );  
 }  
};  
// 相当于私有数据  
var dataPriv = new Data();  
// 用户数据  
var dataUser = new Data();  
  
  
  
// 实现概要  
// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*学习下如何不暴露实现细节给用户  
// 1. 增加api接口，语义上兼容1.9x分支  
// 2. 通过减少存储路径到单个，以提高模块可维护性  
// 3. 使用相同的机制支持private与user数据  
// 4. 不会暴露私有数据给用户代码 (**TODO: Drop \_data, \_removeData)**// 5. 避免暴露实现细节给用户对象 (eg. expando properties)  
// 6. 提供清晰方法来实现2014年的WeakMap升级 https://www.debuggex.com/  
  
// \w :匹配包括下划线的任何单词字符,等价于 [A-Z a-z 0-9\_],  
// \W :匹配任何非单词字符,等价于 [^A-Z a-z 0-9\_]  
var rbrace = /^(?:\{[\w\W]\*\}|\[[\w\W]\*\])$/,  
 rmultiDash = /[A-Z]/g;  
// 对数据进行转换，dataAttr中使用，将html中data-\*的数据进行下处理  
function getData( data ) {  
 if ( data === "true" ) {  
 return true;  
 }  
  
 if ( data === "false" ) {  
 return false;  
 }  
  
 if ( data === "null" ) {  
 return null;  
 }  
  
 // 只有在不改变字符串的情况下才转换成数字。  
 if ( data === +data + "" ) {  
 return +data;  
 }  
 // 在此函数外进行了捕获  
 // /^(?:\{[\w\W]\*\}|\[[\w\W]\*\])$/.test('{asd}');=====>true但JSON.parse会报错？  
 if ( rbrace.test( data ) ) {  
 return JSON.parse( data );  
 }  
  
 return data;  
}  
  
// 从HTML的data-\*获取数据，将数据设置到dataUser里面  
function dataAttr( elem, key, data ) {  
 var name;  
  
 // 如内部找不到任何数据，则从html5中的data-\*查找数据  
 if ( data === undefined && elem.nodeType === 1 ) {  
 // key.replace( rmultiDash, "-$&" )类似于key.replace( /([A-Z])/g, "-$1" );不知这样写有何意义？？？？？？？？？？？？？  
 name = "data-" + key.replace( rmultiDash, "-$&" ).toLowerCase();  
 data = elem.getAttribute( name );  
  
 if ( typeof data === "string" ) {  
 try {  
 data = getData( data );  
 } catch ( e ) {} // 对于getData报错处理  
  
 // 确保我们设置了数据，这样它以后不会改变。  
 dataUser.set( elem, key, data );  
 } else {  
 data = undefined;  
 }  
 }  
 return data;  
}  
// $.data()方法，每次调用都会针对无this.expando创建cache存放缓存数据  
jQuery.extend( {  
 hasData: function( elem ) {  
 return dataUser.hasData( elem ) || dataPriv.hasData( elem );  
 },  
 //data方法可以同时set或get数据  
 data: function( elem, name, data ) {  
 return dataUser.access( elem, name, data );  
 },  
  
 removeData: function( elem, name ) {  
 dataUser.remove( elem, name );  
 },  
  
 // **TODO: Now that all calls to \_data and \_removeData have been replaced** // with direct calls to dataPriv methods, these can be deprecated.  
 \_data: function( elem, name, data ) {  
 return dataPriv.access( elem, name, data );  
 },  
  
 \_removeData: function( elem, name ) {  
 dataPriv.remove( elem, name );  
 }  
} );  
// $('aa').data()方法，与$.data()方法是不一样的  
jQuery.fn.extend( {  
 data: function( key, value ) {  
 var i, name, data,  
 elem = this[ 0 ],// $匹配到的第一个元素  
 attrs = elem && elem.attributes;// 返回指定节点的属性集合  
  
 // 获取所有值,即$().data()调用形式  
 if ( key === undefined ) {  
 if ( this.length ) {// jquery有匹配到元素  
 data = dataUser.get( elem );  
 // 主要是将html属性上的data-数据存储在data中  
 if ( elem.nodeType === 1 && !dataPriv.get( elem, "hasDataAttrs" ) ) {  
 i = attrs.length;  
 while ( i-- ) {  
  
 // Support: IE 11 only  
 // attrs 元素可以为null (#14894)  
 if ( attrs[ i ] ) { // 获取html上data  
 name = attrs[ i ].name;  
 if ( name.indexOf( "data-" ) === 0 ) {  
 name = jQuery.camelCase( name.slice( 5 ) );  
 dataAttr( elem, name, data[ name ] );  
 }  
 }  
 }  
 dataPriv.set( elem, "hasDataAttrs", true );//只需要存储data-上数据一个次  
 }  
 }  
  
 return data;  
 }  
  
 // 设置多组值  
 if ( typeof key === "object" ) {  
 return this.each( function() {  
 dataUser.set( this, key );  
 } );  
 }  
  
 return access( this, function( value ) {  
 var data;  
  
 // 如$()获取到dom，不是空，则this[0]不为空，value这个参数也不会是undefined  
 // 对于空的jquery对象，如$('#a')[0]会返回undefined，即elem = this[0]会在试图读取data缓存时抛出异常  
 if ( elem && value === undefined ) {  
  
 // 尝试从cahce中获取数据  
 // key在Data中都是驼峰标记法  
 data = dataUser.get( elem, key );  
 if ( data !== undefined ) {  
 return data;  
 }  
  
 // 尝试在html的data中获取数据  
 // HTML5 custom data-\* attrs  
 data = dataAttr( elem, key );  
 if ( data !== undefined ) {  
 return data;  
 }  
  
 // 确实无数据，直接return  
 return;  
 }  
  
 // 设置数据  
 this.each( function() {  
  
 // 总是存储驼峰标记法的key  
 dataUser.set( this, key, value );  
 } );  
 }, null, value, arguments.length > 1, null, true );  
 },  
 // 由于each,则每个节点都会删除key对应的值  
 removeData: function( key ) {  
 return this.each( function() {  
 dataUser.remove( this, key );  
 } );  
 }  
} );

# 队列操作

## 概述

* + - 1. Queue队列是animate动画依赖的基础设施,整个jQuery中队列仅供给动画使用

## 源码分析

jQuery.extend( {  
 queue: function( elem, type, data ) {  
 var queue;  
  
 if ( elem ) {  
 type = ( type || "fx" ) + "queue"; // 专职供fx动画队列处理的  
 // queue对应的key默认为fxqueue,而且默认queue是数组  
 queue = dataPriv.get( elem, type );  
  
 // 如仅是查找，则快速出列  
 if ( data ) {  
 if ( !queue || Array.isArray( data ) ) {  
 // 因为queue返回的为数组，故需要将data转为数组，存在dataPrive的type+queue中  
 queue = dataPriv.access( elem, type, jQuery.makeArray( data ) ); // // 将类数组对象转换为数组对象  
 } else {  
 queue.push( data );  
 }  
 }  
 return queue || [];  
 }  
 },  
 /\*\*  
 \* $.queue(body, 'aa', function(){console.log('aa');});  
 \* $.dequeue(body,'aa');//输出aa  
 \* 出列就有点类似shift的操作，但是不同的是还会执行这个cb1与cb2  
 \* 注意：每调用一次仅出列一个  
 \* @param elem  
 \* @param type  
 \*/  
 dequeue: function( elem, type ) {  
 type = type || "fx";  
  
 var queue = jQuery.queue( elem, type ),// 获取elem上的数组  
 startLength = queue.length,  
 fn = queue.shift(), //拿到queue的头元素  
 hooks = jQuery.\_queueHooks( elem, type ),  
 next = function() {  
 jQuery.dequeue( elem, type );  
 };  
  
 // 第二次调用$.dequeue时，上面fn = queue.shift()会将fn赋值为inprogress，进入if后  
 // 将fn赋值为queue实际的函数  
 // ????????????????????????????????????这个操作有何目的？？？？  
 // 英文注释是避免自动出列  
 if ( fn === "inprogress" ) {  
 fn = queue.shift();  
 startLength--;  
 }  
  
 if ( fn ) {  
  
 // 为fx队列增加一个过程标尺，避免自动出列  
 if ( type === "fx" ) {  
 queue.unshift( "inprogress" ); // 向数组添加的第一个元素。  
 }  
  
 // jQuery.fn.delay 会为函数增加一个stop = function(){window.clearTimeout()}  
 // 删除这个属性  
 delete hooks.stop;  
 fn.call( elem, next, hooks );  
 }  
 // 当队列=0时，删除dataPriv的type+queue  
 if ( !startLength && hooks ) {  
 hooks.empty.fire();  
 }  
 },  
  
 // 非公共方法，产生一个queueHooks对象，或返回现在存在的一个  
 // 在当前元素elem的type + "queueHooks"，创建一个{empty：。。。}  
 \_queueHooks: function( elem, type ) {  
 var key = type + "queueHooks";  
 return dataPriv.get( elem, key ) || dataPriv.access( elem, key, {  
 empty: jQuery.Callbacks( "once memory" ).add( function() {  
 dataPriv.remove( elem, [ type + "queue", key ] );  
 } )  
 } );  
 }  
} );  
  
jQuery.fn.extend( {  
 queue: function( type, data ) {  
 var setter = 2;  
  
 if ( typeof type !== "string" ) {  
 data = type;  
 type = "fx";  
 setter--;  
 }  
 // 如$().queue只有一个参数并且是String，则直接调用$.queue  
 if ( arguments.length < setter ) {  
 return jQuery.queue( this[ 0 ], type );//获取匹配元素的一个,在匹配的第一个上面绑定队列  
 }  
  
 return data === undefined ?  
 this : // $().queue('aaa')情况,返回jquery对象  
 this.each( function() {  
 // 此处的this为jquery的匹配到的每个元素  
 var queue = jQuery.queue( this, type, data );  
  
 // 确保每个queue有一个hooks  
 jQuery.\_queueHooks( this, type );  
 // ?????????????????????????这个地方有问题，type应为函数，否则在dequeue中会有fn.call的调用  
 // 会报错  
 if ( type === "fx" && queue[ 0 ] !== "inprogress" ) {  
 jQuery.dequeue( this, type );  
 }  
 } );  
 },  
 dequeue: function( type ) {  
 return this.each( function() {  
 jQuery.dequeue( this, type );  
 } );  
 },  
 // 清除队列，就是将type类型的数据存储置[]  
 clearQueue: function( type ) {  
 return this.queue( type || "fx", [] );  
 },  
  
 // 获得一个promise resolved，当某个队列的type为空时，默认type为fx  
 promise: function( type, obj ) {  
 var tmp,  
 count = 1,  
 defer = jQuery.Deferred(),  
 elements = this,  
 i = this.length,  
 resolve = function() {  
 if ( !( --count ) ) {  
 defer.resolveWith( elements, [ elements ] );  
 }  
 };  
  
 if ( typeof type !== "string" ) {  
 obj = type;  
 type = undefined;  
 }  
 type = type || "fx";  
  
 while ( i-- ) {  
 tmp = dataPriv.get( elements[ i ], type + "queueHooks" );  
 if ( tmp && tmp.empty ) {  
 count++;  
 tmp.empty.add( resolve );  
 }  
 }  
 resolve();  
 return defer.promise( obj );  
 }  
} );

# 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>

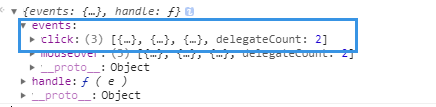
* + - 1. js测试代码

$('#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

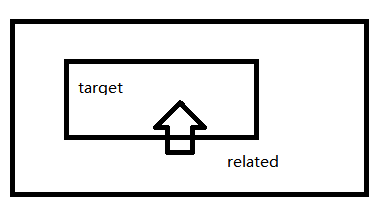
// Support: Firefox <=44  
// 将focus绑定的事件转化为focusin来绑定，focusin在W3C的标准中是冒泡的，除开火狐之外的浏览器也确实支持冒泡  
// firefox不支持focus(in | out)事件- https://bugzilla.mozilla.org/show\_bug.cgi?id=687787  
//  
// Support: Chrome <=48 - 49, Safari <=9.0 - 9.1  
// focus(in | out) 事件在focus & blur events之后触发,  
// 这违背了规范 - http://www.w3.org/TR/DOM-Level-3-Events/#events-focusevent-event-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)

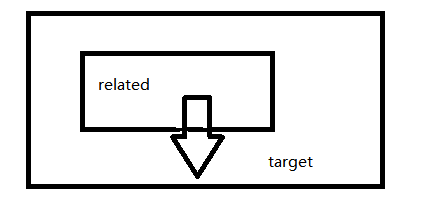
// 用于模拟focus事件  
// 仅仅在`focus(in | out)` 事件上使用，就是对事件e进行改变  
simulate: function( type, elem, event ) {  
 var e = jQuery.extend(  
 new jQuery.Event(),  
 event,  
 {  
 type: type,  
 isSimulated: true  
 }  
 );  
  
 jQuery.event.trigger( e, null, elem );  
}

### 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" );  
 }  
},

# 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

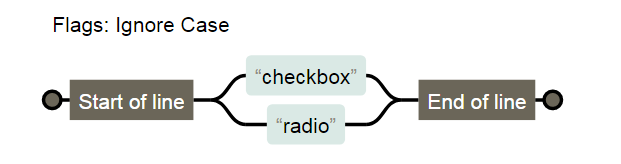
## 克隆节点

### 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)

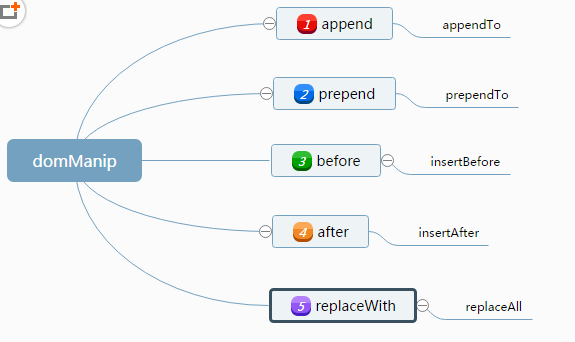
* + - 1. 克隆匹配的元素

/\*\*  
 \* 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。

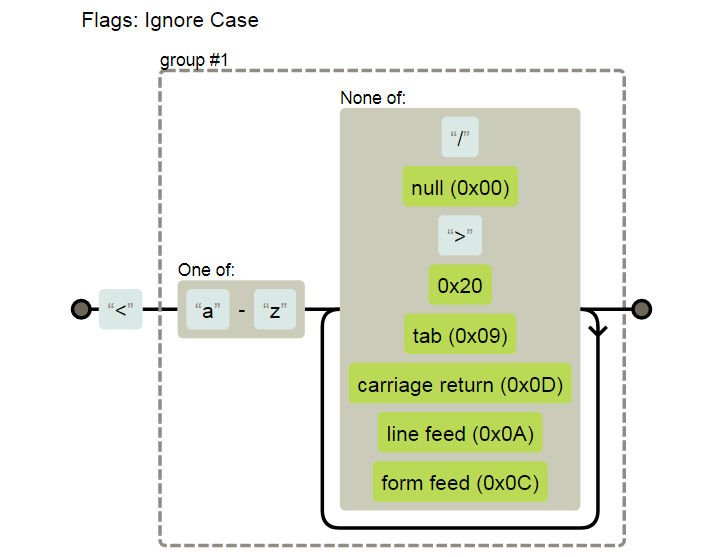
// 删除匹配的元素集合中所有的子节点。  
empty: function() {  
 var elem,  
 i = 0;  
  
 for ( ; ( elem = this[ i ] ) != null; i++ ) {  
 if ( elem.nodeType === 1 ) {  
  
 // 防止内存泄漏  
 jQuery.cleanData( getAll( elem, false ) );  
  
 // 删除所有剩余节点  
 // 一个节点及其后代的文本内容  
 // 在节点上设置 textContent 属性的话，会删除它的所有子节点  
 elem.textContent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

# 选择器

## jQuery.fn.init——line2960

（1）jQuery通过jQuery.fn.init创建jQuery实例

（2）正则表达式理解

rquickExpr = /^(?:\s\*(<[\w\W]+>)[^>]\*|#([\w-]+))$/

(?:) 为非捕获型括号，通常使用它是为了提高性能

\s\*(<[\w\W]+>)[^>]\*

——\s：任何空白字符

——[\w\W]+ : 匹配于'[A-Za-z0-9\_]' 一次或多次

——[^>]\*：字符串尾部是除了>的任意字符或者没有字符

#([\w-]\*))$

——匹配于'[A-Za-z0-9\_- ]与一次或多次

**匹配HTML标记和ID表达式**

## 补：exec和match方法

* + - 1. exec是正则表达式的方法，match是字符串方法
         1. /\d/.exec( "abc4def" );
         2. "abc4def".match(\d);
      2. 两者返回的都是数组，exec没有分组时，如匹配返回只有一个元素的数组
         1. 如不是全局匹配

var str= "cat,hat" ;  
 var p=/at/; //没有g属性  
 console.log(p.exec(str)); //["at", index: 1, input: "cat,hat"]

console.log(str.match(p)); //["at", index: 1, input: "cat,hat"]

* + - * 1. 全局匹配

var str= "cat,hat" ;  
var p=/at/g; //没有g属性  
console.log(p.exec(str));//["at", index: 1, input: "cat,hat"]  
console.log(str.match(p));//["at", "at"]

**exec永远只返回第一个匹配，而match在正则指定了g属性的时候，会返回所有匹配**

* + - 1. 如reg有分组，则exec第一个元素为第一个匹配的字符串，之后为括号的反向引用

var str= "catdtxy2,hatdtxy6" ;  
var p=/c(at)(dt)(x)(y)\d/;  
console.log(p.exec(str));

* + - * 1. [“catdtxy2”,”at”,”dt”,”x”,”y”,index:0,input:”catdtxy2,hatdtxy6”]
      1. match函数在满足如下条件下将越俎代庖，实现和exec一样的功能
         1. 正则表达式中含有分组（括号）
         2. 返回唯一的匹配
      2. 对于(?:)包含的，不会返回到exec数组中

——返回["#id", "#id", undefined, "id", index: 0, input: "#id"]

var b =/^(\s\*(<[\w\W]+>)[^>]\*|#([\w-]+))$/;  
console.log(b.exec("#id"))

* + - * 1. ["#id", undefined, "id", index: 0, input: "#id"]

var b =/^(?:\s\*(<[\w\W]+>)[^>]\*|#([\w-]+))$/;  
console.log(b.exec("#id"))

**大的分组中存在小的分组时，小的分组是排在该大分组后面的分组**