# 什么是Zepto.js

Zepto是一个轻量级的JavaScript库，它的文件大小只有10K左右，兼容现代高级浏览器，主要用于移动端的开发，它有着与jQuery类似的API。就像zepto官网说的如果你会使用jQuery，那你也会用zepto。Zepto有着独特的手指的触摸事件（tap、swipe。。。），且不再支持IE浏览器。

## 下载方式

下载方式：npm install zepto

下载到本地：https://zeptojs.com/zepto.js

中文文档地址：http://www.css88.com/doc/zeptojs\_api/

## 浏览器支持

初级 (100% 支持)

Safari 6+ (Mac)

Chrome 30+ (Windows, Mac, Android, iOS, Linux, Chrome OS)

Firefox 24+ (Windows, Mac, Android, Linux, Firefox OS)

iOS 5+ Safari

Android 2.3+ Browser

Internet Explorer 10+ (Windows, Windows Phone)

次要目标（完全或大部分支持）

iOS 3+ Safari

Chrome <30

Firefox 4+

Safari <6

Android Browser 2.2

Opera 10+

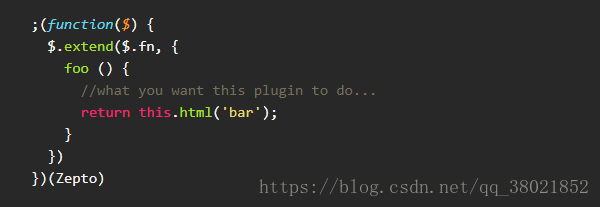
webOS 1.4.5+ Browser

BlackBerry Tablet OS 1.0.7+ Browser

Amazon Silk 1.0+

Other WebKit-based browsers/runtimes

## 创建插件：



## 核心方法：$()

$(selector, [context]) ⇒ collection

$(<Zepto collection>) ⇒ same collection

$(<DOM nodes>) ⇒ collection

$(htmlString) ⇒ collection

$(htmlString, attributes) ⇒ collection v1.0+

Zepto(function($){ ... })

## **原文链接：**

https://blog.csdn.net/qq\_38021852/article/details/82659994

# zepto.js入门到精通（一）

Zepto是一个轻量级的针对现代高级浏览器的JavaScript库， 它与jquery有着类似的api。

但并不是100%覆盖 jQuery 。

Zepto设计的目的是有一个5-10k的通用库、下载并快速执行、有一个熟悉通用的API，

所以你能把你主要的精力放到应用开发上。

用一个script标签引入Zepto到你的页面的底部：

...

<script src=zepto.min.js></script>

</body>

</html>

# [读Zepto源码之操作DOM](https://www.cnblogs.com/libin-1/p/6940382.html)

这篇依然是跟 dom 相关的方法，侧重点是操作 dom 的方法。

读Zepto源码系列文章已经放到了github上，欢迎star: [reading-zepto](https://github.com/yeyuqiudeng/reading-zepto)

## **源码版本**

本文阅读的源码为 [zepto1.2.0](https://github.com/madrobby/zepto/tree/v1.2.0)

## **.remove()**

remove: function() {

return this.each(function() {

if (this.parentNode != null)

this.parentNode.removeChild(this)

})},

删除当前集合中的元素。

如果父节点存在时，则用父节点的 removeChild 方法来删掉当前的元素。

## **相似方法生成器**

zepto 中 after、 prepend、 before、 append、insertAfter、 insertBefore、 appendTo 和 prependTo 都是通过这个相似方法生成器生成的。

### **定义容器**

adjacencyOperators = ['after', 'prepend', 'before', 'append']

首先，定义了一个相似操作的数组，注意数组里面只有 after、 prepend、 before、 append 这几个方法名，后面会看到，在生成这几个方法后，insertAfter、 insertBefore、 appendTo 和 prependTo 会分别调用前面生成的几个方法。

### **辅助方法traverseNode**

function traverseNode(node, fun) {

fun(node)

for (var i = 0, len = node.childNodes.length; i < len; i++)

traverseNode(node.childNodes[i], fun)}

这个方法递归遍历 node 的子节点，将节点交由回调函数 fun 处理。这个辅助方法在后面会用到。

### **核心源码**

adjacencyOperators.forEach(function(operator, operatorIndex) {

var inside = operatorIndex % 2 //=> prepend, append

$.fn[operator] = function() {

// arguments can be nodes, arrays of nodes, Zepto objects and HTML strings

var argType, nodes = $.map(arguments, function(arg) {

var arr = []

argType = type(arg)

if (argType == "array") {

arg.forEach(function(el) {

if (el.nodeType !== undefined) return arr.push(el)

else if ($.zepto.isZ(el)) return arr = arr.concat(el.get())

arr = arr.concat(zepto.fragment(el))

})

return arr

}

return argType == "object" || arg == null ?

arg : zepto.fragment(arg)

}),

parent, copyByClone = this.length > 1

if (nodes.length < 1) return this

return this.each(function(\_, target) {

parent = inside ? target : target.parentNode

// convert all methods to a "before" operation

target = operatorIndex == 0 ? target.nextSibling :

operatorIndex == 1 ? target.firstChild :

operatorIndex == 2 ? target :

null

var parentInDocument = $.contains(document.documentElement, parent)

nodes.forEach(function(node) {

if (copyByClone) node = node.cloneNode(true)

else if (!parent) return $(node).remove()

parent.insertBefore(node, target)

if (parentInDocument) traverseNode(node, function(el) {

if (el.nodeName != null && el.nodeName.toUpperCase() === 'SCRIPT' &&

(!el.type || el.type === 'text/javascript') && !el.src) {

var target = el.ownerDocument ? el.ownerDocument.defaultView : window

target['eval'].call(target, el.innerHTML)

}

})

})

})

}

### **调用方式**

在分析之前，先看看这几个方法的用法：

after(content)prepend(content)before(content)append(content)

参数 content 可以为 html 字符串，dom 节点，或者节点组成的数组。after 是在每个集合元素后插入 content ， before 正好相反，在每个集合元素前插入 content，prepend 是在每个集合元素的初始位置插入 content， append 是在每个集合元素的末尾插入 content。before 和 after 插入的 content 在元素的外部，而 prepend 和 append 插入的 content 在元素的内部，这是需要注意的。

### **将参数 content 转换成 node 节点数组**

var inside = operatorIndex % 2 //=> prepend, append

遍历 adjacencyOperators，得到对应的方法名 operator 和方法名在数组中的索引 operatorIndex。

定义了一个 inside 变量，当 operatorIndex 为偶数时，inside 的值为 true，也就是 operator 的值为 prepend 或 append 时，inside 的值为 true 。这个可以用来区分 content 是插入到元素内部还是外部的方法。

$.fn[operator] 即为 $.fn 对象设置对应的属性值（方法名）。

var argType, nodes = $.map(arguments, function(arg) {

var arr = []

argType = type(arg)

if (argType == "array") {

arg.forEach(function(el) {

if (el.nodeType !== undefined) return arr.push(el)

else if ($.zepto.isZ(el)) return arr = arr.concat(el.get())

arr = arr.concat(zepto.fragment(el))

})

return arr

}

return argType == "object" || arg == null ?

arg : zepto.fragment(arg)}),

变量 argType 用来保存变量变量的类型，也即 content 的类型。nodes 是根据 content 转换后的 node 节点数组。

这里用了 $.map arguments 的方式来获取参数 content ，这里只有一个参数，这什么不用 arguments[0] 来获取呢？这是因为 $.map 可以将数组进行展平，具体的实现看这里《[读zepto源码之工具函数](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%B7%A5%E5%85%B7%E5%87%BD%E6%95%B0.md" \l "map)》。

首先用内部函数 type 来获取参数的类型，关于 type 的实现，在《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "type)》 已经作过分析。

如果参数 content ，也即 arg 的类型为数组时，遍历 arg ，如果数组中的元素存在 nodeType 属性，则断定为 node 节点，就将其 push 进容器 arr 中；如果数组中的元素为 zepto 对象（用 $.zepto.isZ 判断，该方法已经在《[读Zepto源码之神奇的$](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E7%A5%9E%E5%A5%87%E7%9A%84$.md" \l "zeptoisz)》有过分析），不传参调用 get 方法，返回的是一个数组，然后调用数组的 concat 方法合并数组，get 方法在《[读Zepto源码之集合操作](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E9%9B%86%E5%90%88%E6%93%8D%E4%BD%9C.md" \l "get)》有过分析；否则，为 html 字符串，调用 zepto.fragment 处理，并将返回的数组合并，`zepto.fragment 在《[读Zepto源码之神奇的$](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E7%A5%9E%E5%A5%87%E7%9A%84$.md" \l "zeptofragment)》中有过分析。

如果参数类型为 object （即为 zepto 对象）或者 null ，则直接返回。

否则为 html 字符串，调用 zepto.fragment 处理。

parent, copyByClone = this.length > 1if (nodes.length < 1) return this

这里还定义了 parent 变量，用来保存 content 插入的父节点；当集合中元素的数量大于 1 时，变量 copyByClone 的值为 true ，这个变量的作用后面再说。

如果 nodes 的数量比 1 小，也即需要插入的节点为空时，不再作后续的处理，返回 this ，以便可以进行链式操作。

### **用 insertBefore 来模拟所有操作**

return this.each(function(\_, target) {

parent = inside ? target : target.parentNode

// convert all methods to a "before" operation

target = operatorIndex == 0 ? target.nextSibling :

operatorIndex == 1 ? target.firstChild :

operatorIndex == 2 ? target :

null

var parentInDocument = $.contains(document.documentElement, parent)

...})

对集合进行 each 遍历

parent = inside ? target : target.parentNode

如果 node 节点需要插入目标元素 target 的内部，则 parent 设置为目标元素 target，否则设置为当前元素的父元素。

target = operatorIndex == 0 ? target.nextSibling :

operatorIndex == 1 ? target.firstChild :

operatorIndex == 2 ? target :

null

这段是将所有的操作都用 dom 原生方法 insertBefore 来模拟。 如果 operatorIndex == 0 即为 after 时，node 节点应该插入到目标元素 target 的后面，即 target 的下一个兄弟元素的前面；当 operatorIndex == 1 即为 prepend 时，node 节点应该插入到目标元素的开头，即 target 的第一个子元素的前面；当 operatorIndex == 2 即为 before 时，insertBefore 刚好与之对应，即为元素本身。当 insertBefore 的第二个参数为 null 时，insertBefore 会将 node 插入到子节点的末尾，刚好与 append 对应。具体见文档：[Node.insertBefore()](https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore)

var parentInDocument = $.contains(document.documentElement, parent)

调用 $.contains 方法，检测父节点 parent 是否在 document 中。$.contains 方法在《[读zepto源码之工具函数](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%B7%A5%E5%85%B7%E5%87%BD%E6%95%B0.md" \l "contains)》中已有过分析。

### **将 node 节点数组插入到元素中**

nodes.forEach(function(node) {

if (copyByClone) node = node.cloneNode(true)

else if (!parent) return $(node).remove()

parent.insertBefore(node, target)

...})

如果需要复制节点时（即集合元素的数量大于 1 时），用 node 节点方法 cloneNode 来复制节点，参数 true 表示要将节点的子节点和属性等信息也一起复制。为什么集合元素大于 1 时需要复制节点呢？因为 insertBefore 插入的是节点的引用，对集合中所有元素的遍历操作，如果不克隆节点，每个元素所插入的引用都是一样的，最后只会将节点插入到最后一个元素中。

如果父节点不存在，则将 node 删除，不再进行后续操作。

将节点用 insertBefore 方法插入到元素中。

### **处理 script 标签内的脚本**

if (parentInDocument) traverseNode(node, function(el) {

if (el.nodeName != null && el.nodeName.toUpperCase() === 'SCRIPT' &&

(!el.type || el.type === 'text/javascript') && !el.src) {

var target = el.ownerDocument ? el.ownerDocument.defaultView : window

target['eval'].call(target, el.innerHTML)

}})

如果父元素在 document 内，则调用 traverseNode 来处理 node 节点及 node 节点的所有子节点。主要是检测 node 节点或其子节点是否为不指向外部脚本的 script 标签。

el.nodeName != null && el.nodeName.toUpperCase() === 'SCRIPT'

这段用来判断是否为 script 标签，通过 node 的 nodeName 属性是否为 script 来判断。

!el.type || el.type === 'text/javascript'

不存在 type 属性，或者 type 属性为 'text/javascript'。这里表示只处理 javascript，因为 type 属性不一定指定为 text/javascript ，只有指定为 test/javascript 或者为空时，才会按照 javascript 来处理。见[MDN文档script](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script)

!el.src

并且不存在外部脚本。

var target = el.ownerDocument ? el.ownerDocument.defaultView : window

是否存在 ownerDocument 属性，ownerDocument 返回的是元素的根节点，也即 document 对象，document 对象的 defaultView 属性返回的是 document 对象所关联的 window 对象，这里主要是处理 iframe 里的 script，因为在 iframe 中有独立的 window 对象。如果不存在该属性，则默认使用当前的 window 对象。

target['eval'].call(target, el.innerHTML)

最后调用 window 的 eval 方法，执行 script 中的脚本，脚本用 el.innerHTML 取得。

为什么要对 script 元素单独进行这样的处理呢？因为出于安全的考虑，脚本通过 insertBefore 的方法插入到 dom 中时，是不会执行脚本的，所以需要使用 eval 来进行处理。

### **生成 insertAfter、prependTo、insertBefore 和 appendTo 方法**

先来看看这几个方法的调用方式

insertAfter(target)insertBefore(target)appendTo(target)prependTo(target)

这几个方法都是将集合中的元素插入到目标元素 target 中，跟 after、before、append 和 prepend 刚好是相反的操作。

他们的对应关系如下：

after => insertAfter

prepend => prependTo

before => insertBefore

append => appendTo

因此可以调用相应的方法来生成这些方法。

$.fn[inside ? operator + 'To' : 'insert' + (operatorIndex ? 'Before' : 'After')] = function(html) {

$(html)[operator](this)

return this}

inside ? operator + 'To' : 'insert' + (operatorIndex ? 'Before' : 'After')

这段其实是生成方法名，如果是 prepend 或 append ，则在后面拼接 To ，如果是 Before 或 After，则在前面拼接 insert。

$(html)[operator](this)

简单地反向调用对应的方法，就可以了。

到此，这个相似方法生成器生成了after、 prepend、 before、 append、insertAfter、 insertBefore、 appendTo 和 prependTo 等八个方法，相当高效。

## **.empty()**

empty: function() {

return this.each(function() { this.innerHTML = '' })},

empty 的作用是将所有集合元素的内容清空，调用的是 node 的 innerHTML 属性设置为空。

## **.replaceWith()**

replaceWith: function(newContent) {

return this.before(newContent).remove()},

将所有集合元素替换为指定的内容 newContent ， newContent 的类型跟 before 的参数类型一样。

replaceWidth 首先调用 before 将 newContent 插入到对应元素的前面，再将元素删除，这样就达到了替换的上的。

## **.wrapAll()**

wrapAll: function(structure) {

if (this[0]) {

$(this[0]).before(structure = $(structure))

var children

// drill down to the inmost element

while ((children = structure.children()).length) structure = children.first()

$(structure).append(this)

}

return this},

将集合中所有的元素都包裹进指定的结构 structure 中。

如果集合元素存在，即 this[0] 存在，则进行后续操作，否则返回 this ，以进行链式操作。

调用 before 方法，将指定结构插入到第一个集合元素的前面，也即所有集合元素的前面

while ((children = structure.children()).length) structure = children.first()

查找 structure 的子元素，如果子元素存在，则将 structure 赋值为 structure 的第一个子元素，直找到 structrue 最深层的第一个子元素为止。

将集合中所有的元素都插入到 structure 的末尾，如果 structure 存在子元素，则插入到最深层的第一个子元素的末尾。这样就将集合中的所有元素都包裹到 structure 内了。

## **.wrap()**

wrap: function(structure) {

var func = isFunction(structure)

if (this[0] && !func)

var dom = $(structure).get(0),

clone = dom.parentNode || this.length > 1

return this.each(function(index) {

$(this).wrapAll(

func ? structure.call(this, index) :

clone ? dom.cloneNode(true) : dom

)

})},

为集合中每个元素都包裹上指定的结构 structure，structure 可以为单独元素或者嵌套元素，也可以为 html 元素或者 dom 节点，还可以为回调函数，回调函数接收当前元素和当前元素在集合中的索引两个参数，返回符合条件的包裹结构。

var func = isFunction(structure)

判断 structure 是否为函数

if (this[0] && !func)

var dom = $(structure).get(0),

clone = dom.parentNode || this.length > 1

如果集合不为空，并且 structure 不为函数，则将 structure 转换为 node 节点，通过 $(structure).get(0) 来转换，并赋给变量 dom。如果 dom 的 parentNode 存在或者集合的数量大于 1 ，则 clone 的值为 true。

return this.each(function(index) {

$(this).wrapAll(

func ? structure.call(this, index) :

clone ? dom.cloneNode(true) : dom

)})

对集合进行遍历，调用 wrapAll 方法，如果 structure 为函数，则将回调函数返回的结果作为参数传给 wrapAll ；

否则，如果 clone 为 true ，则将 dom 也即包裹元素的副本传给 wrapAll ，否则直接将 dom 传给 wrapAll。这里传递副本的的原因跟生成器中的一样，也是避免对 dom 节点的引用。如果 dom 的 parentNode 存在时，表明 dom 本来就从属于某个节点，如果直接使用 dom ，会破坏原来的结构。

## **.wrapInner()**

wrapInner: function(structure) {

var func = isFunction(structure)

return this.each(function(index) {

var self = $(this),

contents = self.contents(),

dom = func ? structure.call(this, index) : structure

contents.length ? contents.wrapAll(dom) : self.append(dom)

})},

将集合中每个元素的内容都用指定的结构 structure 包裹。 structure 的参数类型跟 wrap 一样。

对集合进行遍历，调用 contents 方法，获取元素的内容，contents 方法在《[读Zepto源码之集合元素查找](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E9%9B%86%E5%90%88%E5%85%83%E7%B4%A0%E6%9F%A5%E6%89%BE.md" \l "contents)》有过分析。

如果 structure 为函数，则将函数返回的结果赋值给 dom ，否则将直接将 structure 赋值给 dom。

如果 contents.length 存在，即元素不为空元素，调用 wrapAll 方法，将元素的内容包裹在 dom 中；如果为空元素，则直接将 dom 插入到元素的末尾，也实现了将 dom 包裹在元素的内部了。

## **.unwrap()**

unwrap: function() {

this.parent().each(function() {

$(this).replaceWith($(this).children())

})

return this},

当集合中的所有元素的包裹层去掉，也即将父元素去掉，但是保留父元素的子元素。

实现的方法也很简单，就是遍历当前元素的父元素，将父元素替换为父元素的子元素。

## **.clone()**

clone: function() {

return this.map(function() { return this.cloneNode(true) })},

每集合中每个元素都创建一个副本，并将副本集合返回。

遍历元素集合，调用 node 的原生方法 cloneNode 创建副本。要注意，cloneNode 不会将元素原来的数据和事件处理程序复制到副本中。

# **读Zepto源码之内部方法**

## **数组方法**

### **定义**

var emptyArray = []

concat = emptyArray.concat

filter = emptyArray.filter

slice = emptyArray.slice

zepto 一开始就定义了一个空数组 emptyArray，定义这个空数组是为了取得数组的 concat、filter、slice 方法

### **compact**

function compact(array) {

return filter.call(array, function(item) {

return item != null

})

}

删除数组中的 null 和 undefined

这里用的是数组的 filter 方法，过滤出 item != null 的元素，组成新的数组。这里删除掉 null 很容易理解，为什么还可以删除 undefined 呢？这是因为这里用了 != ，而不是用 !== ，用 != 时， null 各 undefined 都会先转换成 false 再进行比较。

关于 null 和 undefined 推荐看看这篇文章： [undefined与null的区别](http://www.ruanyifeng.com/blog/2014/03/undefined-vs-null.html)

### **flatten**

function flatten(array) {

return array.length > 0 ? $.fn.concat.apply([], array) : array

}

将数组扁平化，例如将数组 [1,[2,3],[4,5],6,[7,[89]] 变成 [1,2,3,4,5,6,7,[8,9]] ,这个方法只能展开一层，多层嵌套也只能展开一层。

这里，我们先把 $.fn.concat 等价于数组的原生方法 concat，后面的章节也会分析 $.fn.concat 的。

这里比较巧妙的是利用了 apply ，apply 会将 array 中的 item 当成参数，concat.apply([], [1,2,3,[4,5]]) 相当于 [].concat(1,2,3,[4,5])，这样数组就扁平化了。

### **uniq**

uniq = function(array) {

return filter.call(array, function(item, idx) {

return array.indexOf(item) == idx

})

}

数组去重。

数组去重的原理是检测 item 在数组中第一次出现的位置是否和 item 所处的位置相等，如果不相等，则证明不是第一次出现，将其过滤掉。

## **字符串方法**

### **camelize**

camelize = function(str) {

return str.replace(/-+(.)?/g, function(match, chr) {

return chr ? chr.toUpperCase() : ''

})

}

将 word-word 的形式的字符串转换成 wordWord 的形式， - 可以为一个或多个。

正则表达式匹配了一个或多个 - ，捕获组是捕获 - 号后的第一个字母，并将字母变成大写。

### **dasherize**

function dasherize(str) {

return str.replace(/::/g, '/')

.replace(/([A-Z]+)([A-Z][a-z])/g, '$1\_$2')

.replace(/([a-z\d])([A-Z])/g, '$1\_$2')

.replace(/\_/g, '-')

.toLowerCase()

}

将驼峰式的写法转换成连字符 - 的写法。

例如 a = A6DExample::Before

第一个正则表达式是将字符串中的 :: 替换成 / 。a 变成 A6DExample/Before

第二个正则是在出现一次或多次大写字母和出现一次大写字母和连续一次或多次小写字母之间加入 \_。a 变成 A6D\_Example/Before

第三个正则是将出现一次小写字母或数字和出现一次大写字母之间加上 \_。a 变成A6\_D\_Example/Before

第四个正则表达式是将 \_ 替换成 -。a 变成A6-D-Example/Before

最后是将所有的大写字母转换成小写字母。a 变成 a6-d-example/before

我对正则不太熟悉，正则解释部分参考自:[zepto源码--compact、flatten、camelize、dasherize、uniq--学习笔记](http://www.cnblogs.com/zhuhuoxingguang/p/6006743.html)

## **数据类型检测**

### **定义**

class2type = {},

toString = class2type.toString,

// Populate the class2type map$.each("Boolean Number String Function Array Date RegExp Object Error".split(" "), function(i, name) {

class2type["[object " + name + "]"] = name.toLowerCase()

})

$.each 函数后面的文章会讲到，这段代码是将基本类型挂到 class2type 对象上。class2type 将会是如下的形式：

class2type = {

"[object Boolean]": "boolean",

"[object Number]": "number"

...

}

### **type**

function type(obj) {

return obj == null ? String(obj) :

class2type[toString.call(obj)] || "object"

}

type 函数返回的是数据的类型。

如果 obj == null ，也就是 null 和 undefined，返回的是字符串 null 或 undefined

否则调用 Object.prototype.toString （toString = class2type.toString）方法，将返回的结果作为 class2type 的 key 取值。Object.prototype.toString 对不同的数据类型会返回形如 [object Boolean] 的结果。

如果都不是以上情况，默认返回 object 类型。

### **isFunction & isObject**

function isFunction(value) {

return type(value) === 'function'

}function isObject(obj) {

return type(obj) == 'object'

}

调用 type 函数，判断返回的类型字符串，就知道是什么数据类型了

### **isWindow**

function isWindow(obj) {

return obj != null && obj == obj.window

}

判断是否为浏览器的 window 对象

要为 window 对象首先要满足的条件是不能为 null 或者 undefined， 并且 obj.window 为自身的引用。

### **isDocument**

function isDocument(obj) {

return obj != null && obj.nodeType == obj.DOCUMENT\_NODE

}

判断是否为 document 对象

节点上有 nodeType 属性，每个属性值都有对应的常量。document 的 nodeType 值为 9 ，常量为 DOCUMENT\_NODE。

具体见：[MDN文档：Node.nodeType](https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType)

### **isPlainObject**

function isPlainObject(obj) {

return isObject(obj) && !isWindow(obj) && Object.getPrototypeOf(obj) == Object.prototype

}

判断是否为纯粹的对象

纯粹对象首先必须是对象 isObject(obj)

并且不是 window 对象 !isWindow(obj)

并且原型要和 Object 的原型相等

### **isArray**

isArray = Array.isArray ||

function(object) { return object instanceof Array}

这个方法来用判断是否为数组类型。

如果浏览器支持数组的 isArray 原生方法，就采用原生方法，否则检测数据是否为 Array 的实例。

我们都知道，instanceof 的检测的原理是查找构造函数的 prototype 是否在实例的原型链上，如果在，则返回 true。 所以用 instanceof 可能会得到不太准确的结果。例如：

index.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<script> window.onload = function () { var fwindow = window.framePage.contentWindow // frame 页面的window对象 var fArray = fwindow.Array // frame 页面的Array var fdata = fwindow.data // frame 页面的 data [1,2,3] console.log(fdata instanceof fArray) // true console.log(fdata instanceof Array) // false } </script>

<title>Document</title>

</head>

<body>

<iframe id="framePage" src="frame.html" frameborder="0"></iframe>

</body>

</html>

frame.html

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Document</title>

<script> window.data = [1,2,3] </script>

</head>

<body>

<p>frame page</p>

</body>

</html>

由于 iframe 是在独立的环境中运行的，所以 fdata instanceof Array 返回的 false 。

在 MDN 上看到，可以用这样的 ployfill 来使用 isArray

if (!Array.isArray) {

Array.isArray = function(arg) {

return Object.prototype.toString.call(arg) === '[object Array]'

}

}

也就是说，isArray 可以修改成这样：

isArray = Array.isArray ||

function(object) { return Object.prototype.toString.call(object) === '[object Array]'}

为什么 zepto 不这样写呢？知道的可以留言告知下。

### **likeArray**

function likeArray(obj) {

var length = !!obj && // obj必须存在

'length' in obj && // obj 中必须存在 length 属性

obj.length, // 返回 length的值

type = $.type(obj) // 调用 type 函数，返回 obj 的数据类型。这里我有点不太明白，为什么要覆盖掉上面定义的 type 函数呢？再定义多一个变量，直接调用 type 函数不好吗？

return 'function' != type && // 不为function类型

!isWindow(obj) && // 并且不为window类型

(

'array' == type || length === 0 || // 如果为 array 类型或者length 的值为 0，返回true

(typeof length == 'number' && length > 0 && (length - 1) in obj) // 或者 length 为数字，并且 length的值大于零，并且 length - 1 为 obj 的 key

)

}

判断是否为数据是否为类数组。

类数组的形式如下：

likeArrayData = {

'0': 0,

'1': 1,

"2": 2

length: 3

}

可以看到，类数组都有 length 属性，并且 key 为按0,1,2,3 顺序的数字。

代码已经有注释了，这里再简单总结下

首先将 function类型和 window 对象排除

再将 type 为 array 和 length === 0 的认为是类数组。type 为 array 比较容易理解，length === 0 其实就是将其看作为空数组。

最后一种情况必须要满足三个条件：

1. length 必须为数字
2. length 必须大于 0 ，表示有元素存在于类数组中
3. key length - 1 必须存在于 obj 中。我们都知道，数组最后的 index 值为 length -1 ，这里也是检查最后一个 key 是否存在。

# **读zepto源码之工具函数**

## **$.extend**

$.extend 方法可以用来扩展目标对象的属性。目标对象的同名属性会被源对象的属性覆盖。

$.extend 其实调用的是内部方法 extend， 所以我们先看看内部方法 extend 的具体实现。

function extend(target, source, deep) {

for (key in source) // 遍历源对象的属性值

if (deep && (isPlainObject(source[key]) || isArray(source[key]))) { // 如果为深度复制，并且源对象的属性值为纯粹对象或者数组

if (isPlainObject(source[key]) && !isPlainObject(target[key])) // 如果为纯粹对象

target[key] = {} // 如果源对象的属性值为纯粹对象，并且目标对象对应的属性值不为纯粹对象，则将目标对象对应的属性值置为空对象

if (isArray(source[key]) && !isArray(target[key])) // 如果源对象的属性值为数组，并且目标对象对应的属性值不为数组，则将目标对象对应的属性值置为空数组

target[key] = []

extend(target[key], source[key], deep) // 递归调用extend函数

} else if (source[key] !== undefined) target[key] = source[key] // 不对undefined值进行复制

}

extend 的第一个参数 taget 为目标对象， source 为源对象， deep 表示是否为深度复制。当 deep 为 true 时为深度复制， false 时为浅复制。

extend 函数用 for···in 对 source 的属性进行遍历

如果 deep 为 false 时，只进行浅复制，将 source 中不为 undefined 的值赋值到 target 对应的属性中（注意，这里用的是 !==，不是 != ，所以只排除严格为 undefined 的值，不包含 null ）。如果 source 对应的属性值为对象或者数组，会保持该对象或数组的引用。

如果 deep 为 true ，并且 source 的属性值为纯粹对象或者数组时

3.1. 如果 source 的属性为纯粹对象，并且 target 对应的属性不为纯粹对象时，将 target 的对应属性设置为空对象

3.2. 如果 source 的属性为数组，并且 target 对应属性不为数组时，将 target 的对应属性设置为空数组

3.3. 将 source 和 target 对应的属性及 deep 作为参数，递归调用 extend 函数，以实现深度复制。

现在，再看看 $.extend 的具体实现

$.extend = function(target) {

var deep, args = slice.call(arguments, 1)

if (typeof target == 'boolean') {

deep = target

target = args.shift()

}

args.forEach(function(arg) { extend(target, arg, deep) })

return target

}

在说原理之前，先来看看 $.extend 的调用方式，调用方式如下：

$.extend(target, [source, [source2, ...]])

或$.extend(true, target, [source, ...])

在 $.extend 中，如果不需要深度复制，第一个参数可以是目标对象 target, 后面可以有多个 source 源对象。如果需要深度复制，第一个参数为 deep ，第二个参数为 target ，为目标对象，后面可以有多个 source 源对象。

$.extend 函数的参数设计得很优雅，不需要深度复制时，可以不用显式地将 deep 置为 false。这是如何做到的呢？

在 $.extend 函数中，定义了一个数组 args，用来接受除第一个参数外的所有参数。

然后判断第一个参数 target 是否为布尔值，如果为布尔值，表示第一个参数为 deep ，那么第二个才为目标对象，因此需要重新为 target 赋值为 args.shift() 。

最后就比较简单了，循环源对象数组 args， 分别调用 extend 方法，实现对目标对象的扩展。

## **$.each**

$.each 用来遍历数组或者对象，源码如下：

$.each = function(elements, callback) {

var i, key

if (likeArray(elements)) { // 类数组

for (i = 0; i < elements.length; i++)

if (callback.call(elements[i], i, elements[i]) === false) return elements

} else { // 对象

for (key in elements)

if (callback.call(elements[key], key, elements[key]) === false) return elements

}

return elements

}

先来看看调用方式：$.each(collection, function(index, item){ ... })

$.each 接收两个参数，第一个参数 elements 为需要遍历的数组或者对象，第二个 callback 为回调函数。

如果 elements 为数组，用 for 循环，调用 callback ，并且将数组索引 index 和元素值 item 传给回调函数作为参数；如果为对象，用 for···in 遍历属性值，并且将属性 key 及属性值传给回调函数作为参数。

注意回调函数调用了 call 方法，call 的第一个参数为当前元素值或当前属性值，所以回调函数的上下文变成了当前元素值或属性值，也就是说回调函数中的 this 指向的是 item 。这在dom集合的遍历中相当有用。

在遍历的时候，还对回调函数的返回值进行判断，如果回调函数返回 false （if (callback.call(elements[i], i, elements[i]) === false) ） ，立即中断遍历。

$.each 调用结束后，会将遍历的数组或对象（ elements ）返回。

## **$.map**

可以遍历数组（类数组）或对象中的元素，根据回调函数的返回值，将返回值组成一个新的数组，并将该数组扁平化后返回，会将 null 及 undefined 排除。

$.map = function(elements, callback) {

var value, values = [],

i, key

if (likeArray(elements))

for (i = 0; i < elements.length; i++) {

value = callback(elements[i], i)

if (value != null) values.push(value)

}

else

for (key in elements) {

value = callback(elements[key], key)

if (value != null) values.push(value)

}

return flatten(values)

}

先来看看调用方式： $.map(collection, function(item, index){ ... })

elements 为类数组或者对象。callback 为回调函数。当为类数组时，用 for 循环，当为对象时，用 for···in 循环。并且将对应的元素（属性值）及索引（属性名）传递给回调函数，如果回调函数的返回值不为 null 或者 undefined ，则将返回值存入新数组中，最后将新数组扁平化后返回。

## **$.camelCase**

该方法是将字符串转换成驼峰式的字符串

$.camelCase = camelize

$.camelCase 调用的是内部方法 camelize ,该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "camelize)》中已有阐述，本篇文章就不再展开。

## **$.contains**

用来检查给定的父节点中是否包含有给定的子节点，源码如下：

$.contains = document.documentElement.contains ?

function(parent, node) {

return parent !== node && parent.contains(node)

} :

function(parent, node) {

while (node && (node = node.parentNode))

if (node === parent) return true

return false

}

先来看看调用：$.contains(parent, node)

参数 parent 为父子点，node 为子节点。

$.contains 的主体是一个三元表达式，返回的是一个匿名函数。三元表达式的条件是 document.documentElement.contains， 用来检测浏览器是否支持 contains 方法，如果支持，则直接调用 contains 方法，并且将 parent 和 node 为同一个元素的情况排除。

否则，返回另一外匿名函数。该函数会一直向上寻找 node 元素的父元素，如果能找到跟 parent 相等的父元素，则返回 true， 否则返回 false

## **$.grep**

该函数其实就是数组的 filter 函数

$.grep = function(elements, callback) {

return filter.call(elements, callback)

}

从源码中也可以看出，$.grep 调用的就是数组方法 filter

## **$.inArray**

返回指定元素在数组中的索引值

$.inArray = function(elem, array, i) {

return emptyArray.indexOf.call(array, elem, i)

}

先来看看调用 $.inArray(element, array, [fromIndex])

第一个参数 element 为指定的元素，第二个参数为 array 为数组， 第三个参数 fromIndex 为可选参数，表示从哪个索引值开始向后查找。

$.inArray 其实调用的是数组的 indexOf 方法，所以传递的参数跟 indexOf 方法一致。

## **$.isArray**

判断是否为数组

$.isArray = isArray

$.isArray 调用的是内部方法 isArray ，该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "isarray)》中已有阐述。

## **$.isFunction**

判读是否为函数

$.isFunction = isFunction

$.isFunction 调用的是内部方法 isFunction ，该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "isfunction--isobject)》中已有阐述。

## **$.isNumeric**

是否为数值

$.isNumeric = function(val) {

var num = Number(val), // 将参数转换为Number类型

type = typeof val

return val != null &&

type != 'boolean' &&

(type != 'string' || val.length) &&

!isNaN(num) &&

isFinite(num)

|| false

}

判断是否为数值，需要满足以下条件

1. 不为 null
2. 不为布尔值
3. 不为NaN(当传进来的参数不为数值或如'123'这样形式的字符串时，都会转换成NaN)
4. 为有限数值
5. 当传进来的参数为字符串的形式，如'123' 时，会用到下面这个条件来确保字符串为数字的形式，而不是如 123abc这样的形式。(type != 'string' || val.length) && !isNaN(num) 。这个条件的包含逻辑如下：如果为字符串类型，并且为字符串的长度大于零，并且转换成数组后的结果不为NaN，则断定为数值。（因为 Number('') 的值为 0）

## **$.isPlainObject**

是否为纯粹对象，即以 {} 常量或 new Object() 创建的对象

$.isPlainObject = isPlainObject

$.isPlainObject 调用的是内部方法 isPlainObject ，该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "isplainobject)》中已有阐述。

## **$.isWindow**

是否为浏览器的 window 对象

$.isWindow = isWindow

$.isWindow 调用的是内部方法 isWindow ，该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "iswindow)》中已有阐述。

## **$.noop**

空函数

$.noop = function() {}

这个在需要传递回调函数作为参数，但是又不想在回调函数中做任何事情的时候会非常有用，这时，只需要传递一个空函数即可。

## **$.parseJSON**

将标准JSON格式的字符串解释成JSON

if (window.JSON) $.parseJSON = JSON.parse

其实就是调用原生的 JSON.parse， 并且在浏览器不支持的情况下，zepto 还不提供这个方法。

## **$.trim**

删除字符串头尾的空格

$.trim = function(str) {

return str == null ? "" : String.prototype.trim.call(str)

}

如果参数为 null 或者 undefined ，则直接返回空字符串，否则调用字符串原生的 trim 方法去除头尾的空格。

## **$.type**

类型检测

$.type = type

$.type 调用的是内部方法 type ，该方法在前一篇文章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "%E6%95%B0%E6%8D%AE%E7%B1%BB%E5%9E%8B%E6%A3%80%E6%B5%8B)》中已有阐述。

能检测的类型有 "Boolean Number String Function Array Date RegExp Object Error"

# **读Zepto源码之神奇的$**

## **zepto的css选择器 zepto.qsa**

我们都知道，很多时候，我们都用$ 来获取DOM对象，这跟 zepto.qsa 有很大的关系。

### **源码**

zepto.qsa = function(element, selector) {

var found, // 已经找的到DOM

maybeID = selector[0] == '#', // 是否为ID

maybeClass = !maybeID && selector[0] == '.', // 是否为class

nameOnly = maybeID || maybeClass ? selector.slice(1) : selector, // 将id或class前面的符号去掉

isSimple = simpleSelectorRE.test(nameOnly) // 是否为单个选择器

return (element.getElementById && isSimple && maybeID) ?

((found = element.getElementById(nameOnly)) ? [found] : []) :

(element.nodeType !== 1 && element.nodeType !== 9 && element.nodeType !== 11) ? [] :

slice.call(

isSimple && !maybeID && element.getElementsByClassName ?

maybeClass ? element.getElementsByClassName(nameOnly) :

element.getElementsByTagName(selector) :

element.querySelectorAll(selector)

)

}

以上是 qsa 的所有代码，里面有用到一个正则表达式 simpleSelectorRE，先将这个正则消化下。

simpleSelectorRE = /^[\w-]\*$/,

看到这个正则其实是匹配 a-z、A-Z、0-9、下划线、连词符 组合起来的单词，这其实就是单个 id 和 class 的命名规则。

从 return 中可以看出，qsa 其实是根据不同情况分别调用了原生的 getElementById、getElementsByClassName、getElementsByTagName 和 querySelectorAll 的方法。

为什么要这么麻烦，不直接调用 querySelectorAll 方法呢？这是出于性能的考虑。这里有个简单的[测试](http://jsbin.com/domucixawu/edit?html,css,js,console,output)。这个测试里，页面上只有一个元素，如果比较复杂的时候，差距更加明显。

好了，开始逐行分析代码。

### **参数**

* element 开始查找的元素
* selector 选择器

### **变量**

* found： 已经找到的元素
* maybeID = selector[0] == '#'： 判断选择器的第一个字符是否为 #， 如果是 # ，则可能是 id 选择器
* maybeClass = !maybeID && selector[0] == '.' 如果不是 id 选择器，并且选择器的第一个字符为 . ，则可能是 class 选择器
* nameOnly = maybeID || maybeClass ? selector.slice(1) : selector ，如果为 id 选择器或者 class 选择器，则将第一个字符去掉
* isSimple = simpleSelectorRE.test(nameOnly) 是否为单选择器，即 .single 的形式，不是 .first .secend 等形式

### **element.getElementById**

(element.getElementById && isSimple && maybeID) 这是采用 element.getElementById 的条件。

首先要确保 element 具有 getElementById 的方法。getElementById 的方法是在 document 上的，Chrome等浏览器上，element 可能并不具有 geElementById 的方法，具体可以看看这篇文章：[各浏览器对document.getElementById等方法的实现差异解析](http://www.jb51.net/article/44147.htm)

然后要确保选择器为单选择器，并且为 id 选择器。

返回值为 ((found = element.getElementById(nameOnly)) ? [found] : [])， 如果能查找到元素，则将元素以数组的形式返回，否则返回空数组

### **排除不合法的element**

element.nodeType !== 1 && element.nodeType !== 9 && element.nodeType !== 11 。1 对应的是 Node.ELEMENT\_NODE ，9对应的是 Node.DOCUMENT\_NODE ， 11 对应的是 Node.DOCUMENT\_FRAGMENT\_NODE ，如果不为以上三种类型，直接返回 []。

### **终极三元表达式**

slice.call(

isSimple && !maybeID && element.getElementsByClassName ? // 如果为单选择器并且不为id选择器并且存在getElementsByClassName方法，进入下一个三元表达式判断

maybeClass ? element.getElementsByClassName(nameOnly) : // 如果为class选择器，则采用getElementsByClassName

element.getElementsByTagName(selector) : // 否则采用getElementsByTagName方法

element.querySelectorAll(selector) // 以上情况都不是，则用querySelectorAll

)

这里用了 slice.call 处理所获取到的集合，这样，获取到的DOM集合就可以直接使用数组的方法了。

## **zepto.Z 函数**

从第一篇[代码结构](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E4%BB%A3%E7%A0%81%E7%BB%93%E6%9E%84.md)中我们已经知道，其实实现 $ 函数的核心是 zepto.init ，而 zepto.init 最终返回的是 zepto.Z 的结果。那就先来看看 zepto.Z

zepto.Z = function(dom, selector) {

return new Z(dom, selector)

}

zepto.Z 的代码很简单，返回的是 Z 函数的实例。那接下来再看看 Z 函数：

function Z(dom, selector) {

var i, len = dom ? dom.length : 0

for (i = 0; i < len; i++) this[i] = dom[i]

this.length = len

this.selector = selector || ''

}

Z 函数做的事情也很简单，就是将 dom 数组转化为类数组的形式，并设置对应的 length 属性和 selector 属性。

### **zepto.isZ**

zepto.isZ = function(object) {

return object instanceof zepto.Z

}

既然看了 Z 函数，就顺便也将 isZ 也一起看了吧。isZ 函数用来判断参数 object 是否为 Z 的实例，这在 init 中会用到。

## **$的实现 zepto.init 函数**

### **$的实现**

$ = function(selector, context) {

return zepto.init(selector, context)

}

可以看到，其实 $ 调用的就是 zepto.init 这个内部方法。

### **zepto.init**

zepto.init = function(selector, context) {

var dom // dom 集合

if (!selector) return zepto.Z() // 分支1

else if (typeof selector == 'string') { // 分支2

selector = selector.trim()

if (selector[0] == '<' && fragmentRE.test(selector))

dom = zepto.fragment(selector, RegExp.$1, context), selector = null

else if (context !== undefined) return $(context).find(selector)

else dom = zepto.qsa(document, selector)

}

else if (isFunction(selector)) return $(document).ready(selector) // 分支3

else if (zepto.isZ(selector)) return selector // 分支4

else { // 分支5

if (isArray(selector)) dom = compact(selector)

else if (isObject(selector))

dom = [selector], selector = null

else if (fragmentRE.test(selector))

dom = zepto.fragment(selector.trim(), RegExp.$1, context), selector = null

else if (context !== undefined) return $(context).find(selector)

else dom = zepto.qsa(document, selector)

}

return zepto.Z(dom, selector)

}

这个 init 方法代码量不多，但是有大量的 if else， 希望我可以说得清楚

### **$的用法**

$(selector, [context]) ⇒ collection // 用法1$(<Zepto collection>) ⇒ same collection // 用法2$(<DOM nodes>) ⇒ collection // 用法3$(htmlString) ⇒ collection // 用法4$(htmlString, attributes) ⇒ collection v1.0+ // 用法5Zepto(function($){ ... }) // 用法6

### **不传参调用**

直接调用 $() 时，对应的是****分支1****的情况： if (!selector) return zepto.Z() ，返回的是空的 Z 对象

### **selector 为 String 时**

当 selector 为 string 时，对应的代码在****分支2****，对应的用法是****用法1****、****用法4****和****用法5****

在这个分支里，又有三个子分支。一一来看一下：

第一个的判断条件为 selector[0] == '<' && fragmentRE.test(selector) 。selector 的第一个字符为 < ，并且为html标签 。fragmentRE 的定义如下 fragmentRE = /^\s\*<(\w+|!)[^>]\*>/ ，这个其实就是用来判断字符串是否为标签。 我对正则也不太熟，这里就不再展开。

如果满足条件，则执行如下代码：dom = zepto.fragment(selector, RegExp.$1, context), selector = null。 zepto.fragment 其实是通过 htmlString 返回一个dom集合。这个函数稍后会说到，这里先不展开。这里对应的是****用法4****和****用法5****。

如果不满足第一个判断条件，则再判断 context !== undefined (上下文是否存在)。如果存在，则查找 context 下选择器为 selector 的所有子元素: $(context).find(selector) 。这个分支对应的是****用法1****

否则，调用 zepto.qsa 方法，查找 document 下的所有 selector ： dom = zepto.qsa(document, selector)。这里对应的是****用法1****。

### **selector 为 Function 时**

对应的代码在****分支3****，对应的用法是****用法6****

这个分支很简单，在页面加载完毕后，再执行回调方法：$(document).ready(selector)

用过 zepto 的应该都熟悉这种用法: $(function() {})。其实走的就是这个分支

### **selector 为 Z 对象时**

对应的代码在****分支4****，对应的用法是****用法2****

如果参数已经为 Z 对象（zepto.isZ(selector)），则不需要做任何事情，直接原对象返回就可以了。

### **selector 为其他情况**

如果为数组时（isArray(selector)）, 将数组展平(dom = compact(selector))

如果为对象时（isObject(selector)），将对象包裹成数组（dom = [selector]）。

以上两种情况对应的是****用法3****，将dom对象或dom集合转化为 z 对象

如果为标签（fragmentRE.test(selector)），执行跟****分支1****一模一样的代码。这里判断在上面已经做过了，为什么要再来一次呢？我也不太明白，有明白的可以跟我说下。

经过一轮又一轮的判断和 selector 重置，现在终于可以调用 z 函数了: zepto.Z(dom, selector) ，init 的最后，将收集到的 dom 集合和对应的 selector 传入 Z 函数，返回 Z 对象。

## **zepto.fragment**

zepto.fragment = function(html, name, properties) {

var dom, nodes, container

if (singleTagRE.test(html)) dom = $(document.createElement(RegExp.$1))

if (!dom) {

if (html.replace) html = html.replace(tagExpanderRE, "<$1></$2>")

if (name === undefined) name = fragmentRE.test(html) && RegExp.$1

if (!(name in containers)) name = '\*'

container = containers[name]

container.innerHTML = '' + html

dom = $.each(slice.call(container.childNodes), function() {

container.removeChild(this)

})

}

if (isPlainObject(properties)) {

nodes = $(dom)

$.each(properties, function(key, value) {

if (methodAttributes.indexOf(key) > -1) nodes[key](value)

else nodes.attr(key, value)

})

}

return dom

}

fragment 的作用的是将html片断转换成dom数组形式。

首先判断是否为标签的形式 singleTagRE.test(html) （如<div></div>）, 如果是，则采用该标签名来创建dom对象 dom = $(document.createElement(RegExp.$1))，不用再作其他处理。singleTagRE = /^<(\w+)\s\*\/?>(?:<\/\1>|)$/。

如果尚未获取到 dom，接着进行：

if (html.replace) html = html.replace(tagExpanderRE, "<$1></$2>")

这段是对 html 进行修复，如<p class="test" /> 修复成 <p class="test" /></p> 。正则表达式为 tagExpanderRE = /<(?!area|br|col|embed|hr|img|input|link|meta|param)(([\w:]+)[^>]\*)\/>/ig

if (name === undefined) name = fragmentRE.test(html) && RegExp.$1

如果没有指定标签名，则获取标签名。如传入 <div>test</div> ，获取到的 name 为 div

if (!(name in containers)) name = '\*'

container = containers[name]

container.innerHTML = '' + html

dom = $.each(slice.call(container.childNodes), function() {

container.removeChild(this)

})

}// containers 已经开头定义，如下

table = document.createElement('table'),

tableRow = document.createElement('tr'),

containers = {

'tr': document.createElement('tbody'),

'tbody': table,

'thead': table,

'tfoot': table,

'td': tableRow,

'th': tableRow,

'\*': document.createElement('div')

}

检测 name 是否为特殊的元素，如 tr 要用 tbody 包裹，其他的元素用 div 包裹。包裹元素的 childNodes 即为所需要获取的 dom 。

if (isPlainObject(properties)) {

nodes = $(dom)

$.each(properties, function(key, value) {

if (methodAttributes.indexOf(key) > -1) nodes[key](value)

else nodes.attr(key, value)

})

}// methodAttributes 在上面已经定义，定义如下

methodAttributes = ['val', 'css', 'html', 'text', 'data', 'width', 'height', 'offset']

如果属性值为纯对象，则给元素设置属性。

如果所需设置的属性，zepto已经定义了相应的方法，则调用zepto对应的方法，否则统一调用zepto的attr 方法设置属性。

最后将 dom 返回

# **读Zepto源码之集合操作**

## **.forEach()**

forEach: emptyArray.forEach

因为 zepto 的 dom 集合是类数组，所以这里只是简单地复制了数组的 forEach 方法。

具体的 forEach 的用法见文档:[Array.prototype.forEach()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach?v=example)

## **.reduce()**

reduce: emptyArray.reduce

简单地复制了数组的 reduce 方法。

具体的 reduce 的用法见文档:[Array.prototype.reduce()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce?v=example)

## **.push()**

push: emptyArray.push

简单地复制了数组的 push 方法。

具体的 push 的用法见文档:[Array.prototype.push()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push?v=example)

## **.sort()**

sort: emptyArray.sort

简单地复制了数组的 sort 方法。

具体的 sort 的用法见文档:[Array.prototype.sort()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/sort?v=example)

## **.splice()**

splice: emptyArray.splice

简单地复制了数组的 splice 方法。

具体的 splice 的用法见文档:[Array.prototype.splice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice?v=example)

## **.indexOf()**

indexOf: emptyArray.indexOf

简单地复制了数组的 indexOf 方法。

具体的 indexOf 的用法见文档:[Array.prototype.indexOf()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf)

## **.get()**

get: function(idx) {

return idx === undefined ? slice.call(this) : this[idx >= 0 ? idx : idx + this.length]

},

这个方法用来获取指定索引值的元素。

不传参（idx === undefined）时，不传参调用数组的 slice 方法，将集合中的所有元素返回。

当传递的参数大于或等于零（idx）时，返回相应索引值的元素 this[idx] ，如果为负数，则倒数返回this.[idx + this.length]。

例如 $('li').get(-1) 返回的是倒数第1个元素，也即最后一个元素

## **.toArray()**

toArray: function() { return this.get() }

toArray 方法是将元素的类数组变成纯数组。toArray 内部不传参调用 get 方法，上面已经分析了，当不传参数时，get 方法调用的是数组方法 slice， 返回的自然就是纯数组了。

## **.size()**

size: function() {

return this.length

}

size 方法返回的是集合中的 length 属性，也即集合中元素的个数。

## **.concat()**

concat: function() {

var i, value, args = []

for (i = 0; i < arguments.length; i++) {

value = arguments[i]

args[i] = zepto.isZ(value) ? value.toArray() : value

}

return concat.apply(zepto.isZ(this) ? this.toArray() : this, args)

},

数组中也有对应的 concat 方法，为什么不能像上面的方法那样直接调用呢？

这是因为 $.fn 其实是一个类数组对象，并不是真正的数组，如果直接调用 concat 会直接把整个 $.fn 当成数组的一个 item 合并到数组中。

for (i = 0; i < arguments.length; i++) {

value = arguments[i]

args[i] = zepto.isZ(value) ? value.toArray() : value

}

这段是对每个参数进行判断，如果参数是 zepto 的集合（zepto.isZ(value)），就先调用 toArray 方法，转换成纯数组。

return concat.apply(zepto.isZ(this) ? this.toArray() : this, args)

这段同样对 this 进行了判断，如果为 zepto 集合，也先转换成数组。所以调用 concat 后返回的是纯数组，不再是 zepto 集合。

## **.map()**

map: function(fn) {

return $($.map(this, function(el, i) { return fn.call(el, i, el) }))

}

map 方法的内部调用的是 zepto 的工具函数 $.map ，这在之前已经在《[读Zepto源码之工具函数](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%B7%A5%E5%85%B7%E5%87%BD%E6%95%B0.md" \l "map)》做过了分析。

return fn.call(el, i, el)

map 方法对回调也做了包装，call 的第一个参数为 el ，因此可以在 map 的回调中通过 this 来拿到每个元素。

map 方法对 $.map 返回的数组调用了 $() 方法，将返回的数组再次包装成 zepto 对象，因此调用 map 方法后得到的数组，同样具有 zepto 集合中的方法。

## **.slice()**

slice: function() {

return $(slice.apply(this, arguments))

}

slice 同样没有直接用数组的原生方法，也像 map 方法一样，将返回的数组再次包装成 zepto 对象。

## **.each()**

each: function(callback) {

emptyArray.every.call(this, function(el, idx) {

return callback.call(el, idx, el) !== false

})

return this

},

zepto 的 each 方法比较巧妙，在方法内部，调用的其实是数组的 every 方法，every 遇到 false 时就会中止遍历，zepto 也正是利用 every 这种特性，让 each 方法也具有了中止遍历的能力，当 callback 返回的值为布尔值 false 时，中止遍历，注意这里用了 !==，因为 callback 如果没有返回值时，得到的值会是 undefined ，这种情况是需要排除的。

同样，each 的回调中也是可以用 this 拿到每个元素的。

注意，each 方法最后返回的是 this， 所以在 each 调用完后，还可以继续调用 集合中的其他方法，这就是 zepto 的链式调用，这个跟 map 方法中返回 zepto 集合的原理差不多，只不过 each 返回的是跟原来一样的集合，map 方法返回的是映射后的集合。

## **.add()**

add: function(selector, context) {

return $(uniq(this.concat($(selector, context))))

}

add 可以传递两个参数，selector 和 context ，即选择器和上下文。

add 调用 $(selector, context) 来获取符合条件的集合元素，这在上篇文章《[读Zepto源码之神奇的$](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E7%A5%9E%E5%A5%87%E7%9A%84$.md" \l "%E7%9A%84%E5%AE%9E%E7%8E%B0-zeptoinit-%E5%87%BD%E6%95%B0)》已经有详细的论述。

然后调用 concat 方法来合并两个集合，用内部方法 uniq 来过滤掉重复的项，uniq 方法在《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "uniq)》已经有论述。最后也是返回一个 zepto 集合。

# **读Zepto源码之集合元素查找**

## **内部方法**

之前有一章《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md)》是专门解读 zepto 中没有提供给外部使用的内部方法的，但是有几个涉及到 dom的方法没有解读，这里先将本章用到的方法解读一下。

### **matches**

zepto.matches = function(element, selector) {

if (!selector || !element || element.nodeType !== 1) return false

var matchesSelector = element.matches || element.webkitMatchesSelector ||

element.mozMatchesSelector || element.oMatchesSelector ||

element.matchesSelector

if (matchesSelector) return matchesSelector.call(element, selector)

// fall back to performing a selector:

var match, parent = element.parentNode,

temp = !parent

if (temp)(parent = tempParent).appendChild(element)

match = ~zepto.qsa(parent, selector).indexOf(element)

temp && tempParent.removeChild(element)

return match

}

matches 方法用于检测元素( element )是否匹配特定的选择器( selector )。

浏览器也有原生的 matches 方法，但是要到IE9之后才支持。具体见文档：[Element.matches()](https://developer.mozilla.org/en-US/docs/Web/API/Element/matches)

if (!selector || !element || element.nodeType !== 1) return false

这段是确保 selector 和 element 两个参数都有传递，并且 element 参数的 nodeType 为 ELEMENT\_NODE ，如何条件不符合，返回 false

var matchesSelector = element.matches || element.webkitMatchesSelector ||

element.mozMatchesSelector || element.oMatchesSelector ||

element.matchesSelector

if (matchesSelector) return matchesSelector.call(element, selector)

这段是检测浏览器是否原生支持 matches 方法，或者支持带私有前缀的 matches 方法，如果支持，调用原生的 matches，并将结果返回。

var match, parent = element.parentNode,

temp = !parentif (temp)(parent = tempParent).appendChild(element)

match = ~zepto.qsa(parent, selector).indexOf(element)

temp && tempParent.removeChild(element)

return match

如果原生的方法不支持，则回退到用选择器的方法来检测。

这里定义了三个变量，其中 parent 用来存放 element 的父节点， temp 用来判断 element 是否有父元素。值为 temp = !parent ，如果 element 存在父元素，则 temp 的值为 false 。

首先判断是否存在父元素，如果父元素不存在，则 parent = tempParent ，tempParent 已经由一个全局变量来定义，为 tempParent = document.createElement('div') ，其实就是一个 div 空节点。然后将 element 插入到空节点中。

然后，查找 parent 中所有符合选择器 selector 的元素集合，再找出当前元素 element 在集合中的索引。

zepto.qsa(parent, selector).indexOf(element)

再对索引进行取反操作，这样索引值为 0 的值就变成了 -1，是 -1 的返回的是 0，这样就确保了跟 matches 的表现一致。

其实我有点不太懂的是，为什么不跟原生一样，返回 boolean 类型的值呢？明明通过 zepto.qsa(parent, selector).indexOf(element) > -1 就可以做到了，接口表现一致不是更好吗？

最后还有一步清理操作：

temp && tempParent.removeChild(element)

将空接点的子元素清理点，避免污染。

### **children**

function children(element) {

return 'children' in element ?

slice.call(element.children) :

$.map(element.childNodes, function(node) { if (node.nodeType == 1) return node })

}

children 方法返回的是 element 的子元素集合。

浏览器也有原生支持元素 children 属性，也要到IE9以上才支持，见文档[ParentNode.children](https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/children)

如果检测到浏览器不支持，则降级用 $.map 方法，获取 element 的 childNodes 中 nodeType 为 ELEMENT\_NODE 的节点。因为 children 返回的只是元素节点，但是 childNodes 返回的除元素节点外，还包含文本节点、属性等。

这里用到的 $.map 跟数组的原生方法 map 表现有区别，关于 $.map 的具体实现，已经在《[读zepto源码之工具函数](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%B7%A5%E5%85%B7%E5%87%BD%E6%95%B0.md" \l "map)》解读过了。

### **filtered**

function filtered(nodes, selector) {

return selector == null ? $(nodes) : $(nodes).filter(selector)

}

将匹配指定选择器的元素从集合中过滤出来。

如果没有指定 selector ，则将集合包裹成 zepto 对象全部返回，否则调用 filter 方法，过滤出符合条件的元素返回。filter 方法下面马上讲到。

## **元素方法**

这里的方法都是 $.fn 中提供的方法。

### **.filter()**

filter: function(selector) {

if (isFunction(selector)) return this.not(this.not(selector))

return $(filter.call(this, function(element) {

return zepto.matches(element, selector)

}))

}

filter 是查找符合条件的元素集合。

参数 selector 可以为 Function 或者选择器，当为 Function 时，调用的其实调用了两次 not 方法，负负得正。关于 not 方法，下面马上会看到。

当为一般的选择器时，调用的是filter 方法，filter 的回调函数调用了 matches ，将符合 selector 的元素返回，并包装成 zepto 对象返回。

### **.not()**

not: function(selector) {

var nodes = []

if (isFunction(selector) && selector.call !== undefined)

this.each(function(idx) {

if (!selector.call(this, idx)) nodes.push(this)

})

else {

var excludes = typeof selector == 'string' ? this.filter(selector) :

(likeArray(selector) && isFunction(selector.item)) ? slice.call(selector) : $(selector)

this.forEach(function(el) {

if (excludes.indexOf(el) < 0) nodes.push(el)

})

}

return $(nodes)

}

not 方法是将集合中不符合条件的元素查找出来。

not 方法的方法有三种调用方式：

not(selector) ⇒ collectionnot(collection) ⇒ collectionnot(function(index){ ... }) ⇒ collection

当 selector 为 Function ，并且有 call 方法时（isFunction(selector) && selector.call !== undefined），相关的代码如下：

this.each(function(idx) {

if (!selector.call(this, idx)) nodes.push(this)

})

调用 each 方法，并且在 selector 函数中，可以访问到当前的元素和元素的索引。如果 selector 函数的值取反后为 true，则将相应的元素放入 nodes 数组中。

当 selector 不为 Function 时， 定义了一个变量 excludes ，这个变量来用接收需要排除的元素集合。接下来又是一串三元表达式（zepto的特色啊）

typeof selector == 'string' ? this.filter(selector)

当 selector 为 string 时，调用 filter ，找出所有需要排除的元素

(likeArray(selector) && isFunction(selector.item)) ? slice.call(selector) : $(selector)

这段我刚开始看时，有点困惑，主要是不明白 isFunction(selector.item) 这个判断条件，后来查了MDN文档[HTMLCollection.item](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCollection/item)，才明白 item 是 HTMLCollection 的一个方法，这个三元表达式的意思是，如果是 HTMLCollection ，则调用 slice.call 得到一个纯数组，否则返回 zepto 对象。

this.forEach(function(el) {

if (excludes.indexOf(el) < 0) nodes.push(el)

})

遍历集合，如果元素不在需要排除的元素集合中，将该元素 push 进 nodes 中。

not 方法最终返回的也是 zepto 对象。

### **.is()**

is: function(selector) {

return this.length > 0 && zepto.matches(this[0], selector)

}

判断集合中的第一个元素是否匹配指定的选择器。

代码也比较简单了，选判断集合不为空，再调用 matches 看第一个元素是否匹配。

### **.find()**

find: function(selector) {

var result, $this = this

if (!selector) result = $()

else if (typeof selector == 'object')

result = $(selector).filter(function() {

var node = this

return emptyArray.some.call($this, function(parent) {

return $.contains(parent, node)

})

})

else if (this.length == 1) result = $(zepto.qsa(this[0], selector))

else result = this.map(function() { return zepto.qsa(this, selector) })

return result

}

find 是查找集合中符合选择器的所有后代元素，如果给定的是 zepto 对象或者 dom 元素，则只有他们在当前的集合中时，才返回。

fid 有三种调用方式，如下：

find(selector) ⇒ collectionfind(collection) ⇒ collectionfind(element) ⇒ collection

if (!selector) result = $()

如果不传参时，返回的是空的 zepto 对象。

else if (typeof selector == 'object')

result = $(selector).filter(function() {

var node = this

return emptyArray.some.call($this, function(parent) {

return $.contains(parent, node)

})

})

如果传参为 object 时，也就是 zepto 对象collection 和dom 节点 element 时，先将 selector 包裹成 zepto 对象，然后对这个对象过滤，返回当前集合子节点中所包含的元素（$.contains(parent, node)）。

else if (this.length == 1) result = $(zepto.qsa(this[0], selector))

如果当前的集合只有一个元素时，直接调用 zepto.qsa 方法，取出集合的第一个元素 this[0] 作为 qsa 的第一个参数。关于 qsa 方法，已经在《[读Zepto源码之神奇的$](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E7%A5%9E%E5%A5%87%E7%9A%84$.md" \l "zepto%E7%9A%84css%E9%80%89%E6%8B%A9%E5%99%A8-zeptoqsa)》分析过了。其实就是获取第一个元素的所有后代元素。

else result = this.map(function() { return zepto.qsa(this, selector) })

否则，调用 map 方法，对集合中每个元素都调用 qsa 方法，获取所有元素的后代元素。这个条件其实可以与上一个条件合并的，分开应该是为了性能的考量。

### **.has()**

has: function(selector) {

return this.filter(function() {

return isObject(selector) ?

$.contains(this, selector) :

$(this).find(selector).size()

})

},

判断集合中是否有包含指定条件的子元素，将符合条件的元素返回。

有两种调用方式

has(selector) ⇒ collectionhas(node) ⇒ collection

参数可以为选择器或者节点。

has 其实调用的是 filter 方法，这个方法上面已经解读过了。filter 的回调函数中根据参数的不同情况，调用了不同的方法。

isObject(selector) 用来判断 selector 是否为 node 节点，如果为 node 节点，则调用 $.contains 方法，该方法已经在《[读Zepto源码之工具函数](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%B7%A5%E5%85%B7%E5%87%BD%E6%95%B0.md" \l "contains)》说过了。

如果为选择器，则调用 find 方法，然后再调用 size 方法，size 方法返回的是集合中元素的个数。这个在《[读Zepto源码之集合操作](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E9%9B%86%E5%90%88%E6%93%8D%E4%BD%9C.md" \l "size)》有讲过，如果集合个数大于零，则表示满足条件。

### **.eq()**

eq: function(idx) {

return idx === -1 ? this.slice(idx) : this.slice(idx, +idx + 1)

},

获取集合中指定的元素。

这里调用了 slice 方法，这个方法在上一篇《[读Zepto源码之集合操作](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E9%9B%86%E5%90%88%E6%93%8D%E4%BD%9C.md" \l "slice)》已经说过了。如果 idx 为 -1 时，直接调用 this.slice(idx) ，即取出最后一个元素，否则取 idx 至 idx + 1 之间的元素，也就是每次只取一个元素。+idx+1 前面的 + 号其实是类型转换，确保 idx 在做加法的时候为 Number 类型。

### **.first()**

first: function() {

var el = this[0]

return el && !isObject(el) ? el : $(el)

},

first 是取集合中第一个元素，这个方法很简单，用索引 0 就可以取出来了，也就是 this[0] 。

上面已经实现了一个 eq 的方法，这里为什么不用 eq 呢？应该是出于性能的考量。eq 内部是用 slice 实现的，我做了个简单的测试，一个长度为100的数组，循环10万次，直接用索引值的时间为1ms，用 slice 是24ms。测试地址：<https://jsfiddle.net/57fk5ayf/>

el && !isObject(el) 用来判断是否为 zepto 对象，如果不是，用 $(el) 包裹，确保返回的是 zepto 对象。这里主要要区分的是 node 节点。在《[读Zepto源码之内部方法](https://github.com/yeyuqiudeng/reading-zepto/blob/master/src/%E8%AF%BBZepto%E6%BA%90%E7%A0%81%E4%B9%8B%E5%86%85%E9%83%A8%E6%96%B9%E6%B3%95.md" \l "isfunction--isobject)》中，我们知道，isObject 最终是通过调用 toString 方法实现的。如果是 node 节点， toString 的结果是 [object HTMLDivElement] ，如果为 zepto 对象，返回的结果是 [object Object] 。因此为 node 节点时，isObject 返回的结果为 false。

### **.last()**

last: function() {

var el = this[this.length - 1]

return el && !isObject(el) ? el : $(el)

},

last 是取集合中最后一个元素，这个的原理跟 first 一样，只不过变成了取索引值为 this.length - 1 ，也就是最后的元素。

### **.closest()**

closest: function(selector, context) {

var nodes = [],

collection = typeof selector == 'object' && $(selector)

this.each(function(\_, node) {

while (node && !(collection ? collection.indexOf(node) >= 0 : zepto.matches(node, selector)))

node = node !== context && !isDocument(node) && node.parentNode

if (node && nodes.indexOf(node) < 0) nodes.push(node)

})

return $(nodes)

},

从元素本身向上查找，返回最先符合条件的元素。

这个方法也有三种调用方式

closest(selector, [context]) ⇒ collectionclosest(collection) ⇒ collection closest(element) ⇒ collection

如果指定了 zepto 集合或者 element ，则只返回匹配给定集合或 element 的元素。

collection = typeof selector == 'object' && $(selector)

这段是判断 selector 是否为 collection 或 element ，如果是，则统一转化为 zepto 集合。

然后对集合遍历，在 each 遍历里针对集合中每个 node 节点，都用 while 语句，向上查找符合条件的元素。

node && !(collection ? collection.indexOf(node) >= 0 : zepto.matches(node, selector))

这段是 while 语句的终止条件。 node 节点必须存在，如果 selector 为 zepto 集合或者 element ，也即 collection存在， 则要找到存在于 collection 中的节点（collection.indexOf(node) >= 0）， 否则，节点要匹配指定的选择器（zepto.matches(node, selector)）

在 while 循环中，是向上逐级查找节点的过程：

node = node !== context && !isDocument(node) && node.parentNode

当前 node 不为指定的上下文 context 并且不为 document 节点时，向上查找（node.parentNode）

if (node && nodes.indexOf(node) < 0) nodes.push(node)

while 循环完毕后，如果 node 节点存在，并且 nodes 中还不存在 node ，则将 node push 进 nodes 中。

最后返回 zepto 集合。

### **.pluck()**

pluck: function(property) {

return $.map(this, function(el) { return el[property] })

},

返回集合中所有元素指定的属性值。

这个方法很简单，就是对当前集合遍历，然后取元素指定的 property 值。

### **.parents()**

parents: function(selector) {

var ancestors = [],

nodes = this

while (nodes.length > 0)

nodes = $.map(nodes, function(node) {

if ((node = node.parentNode) && !isDocument(node) && ancestors.indexOf(node) < 0) {

ancestors.push(node)

return node

}

})

return filtered(ancestors, selector)

},

返回集合中所有元素的所有祖先元素。

nodes 的初始值为当前集合，while 循环的条件为集合不为空。

使用 map 遍历 nodes ，将 node 重新赋值为自身的父级元素，如果父级元素存在，并且不是 document 元素，而且还不存在于 ancestors 中时，将 node 存入保存祖先元素的 ancestors 中，并且 map 回调的返回值是 node ，组成新的集合赋值给 nodes ，直到所有的祖先元素遍历完毕，就可以退出 while 循环。

最后，调用上面说到的 filtered 方法，找到符合 selector 的祖先元素。

### **.parent()**

parent: function(selector) {

return filtered(uniq(this.pluck('parentNode')), selector)

},

返回集合中所有元素的父级元素。

parents 返回的是所有祖先元素，而 parent 返回只是父级元素。

首先调用的是 this.pluck('parentNode') ，获取所有元素的祖先元素，然后调用 uniq 对集合去重，最后调用 filtered，返回匹配 selector 的元素集合。

### **.children()**

children: function(selector) {

return filtered(this.map(function() { return children(this) }), selector)

},

返回集合中所有元素的子元素。

首先对当前集合遍历，调用内部方法 children 获取当前元素的子元素组成新的数组，再调用 filtered 方法返回匹配 selector 的元素集合。

### **.contents()**

contents: function() {

return this.map(function() { return this.contentDocument || slice.call(this.childNodes) })

},

这个方法类似于 children ，不过 children 对 childNodes 进行了过滤，只返回元素节点。contents 还返回文本节点和注释节点。也返回 iframe 的 contentDocument

### **.siblings()**

siblings: function(selector) {

return filtered(this.map(function(i, el) {

return filter.call(children(el.parentNode), function(child) { return child !== el })

}), selector)

},

获取所有集合中所有元素的兄弟节点。

获取兄弟节点的思路也很简单，对当前集合遍历，找到当前元素的父元素el.parentNode，调用 children 方法，找出父元素的子元素，将子元素中与当前元素不相等的元素过滤出来即是其兄弟元素了。

最后调用 filtered 来过滤出匹配 selector 的兄弟元素。

### **.prev()**

prev: function(selector) { return $(this.pluck('previousElementSibling')).filter(selector || '\*') },

获取集合中每个元素的前一个兄弟节点。

这个方法也很简单，调用 pluck 方法，获取元素的 previousElementSibling 属性，即为元素的前一个兄弟节点。再调用 filter 返回匹配 selector 的元素，最后包裹成 zepto 对象返回

### **.next()**

next: function(selector) { return $(this.pluck('nextElementSibling')).filter(selector || '\*') },

next 方法跟 prev 方法类似，只不过取的是 nextElementSibling 属性，获取的是每个元素的下一个兄弟节点。

### **.index()**

index: function(element) {

return element ? this.indexOf($(element)[0]) : this.parent().children().indexOf(this[0])

},

返回指定元素在当前集合中的位置（this.indexOf($(element)[0])），如果没有给出 element ，则返回当前鲜红在兄弟元素中的位置。this.parent().children() 查找的是兄弟元素。