



阿里巴巴资深前端开发工程师撰写，透视jQuery 17个模块的架构设计理念和内部实现原理
全面系统地解读最新版本jQuery源码，帮助读者掌握其中的实现技巧和技术精髓



jQuery Internals

Architecture and Implementation in Depth

jQuery技术内幕

深入解析jQuery架构设计与实现原理



高云 著



jQuery 技术内幕：深入解析 jQuery 架构设计与实现原理



图书在版编目 (CIP) 数据

jQuery 技术内幕：深入解析 jQuery 架构设计与实现原理 / 高云著 . —北京：机械工业出版社，2013.11

ISBN 978-7-111-44082-6

I. j… II. 高… III. JAVA 语言—程序设计 IV. TP312

中国版本图书馆 CIP 数据核字 (2013) 第 221662 号

版权所有·侵权必究

封底无防伪标识均为盗版

本书法律顾问 北京市展达律师事务所

本书由阿里巴巴资深前端开发工程师撰写，从源代码角度全面而系统地解读了 jQuery 的 17 个模块的架构设计理念和内部实现原理，旨在帮助读者参透 jQuery 中的实现技巧和技术精髓，同时本书也对广大开发者如何通过阅读源代码来提升编码能力和软件架构能力提供了指导。

本书首先通过“总体架构”梳理了各个模块的分类、功能和依赖关系，让大家对 jQuery 的工作原理有大致的印象；进而通过“构造 jQuery 对象”章节分析了构造函数 jQuery() 的各种用法和内部构造过程；接着详细分析了底层支持模块的源码实现，包括：选择器 Sizzle、异步队列 Deferred、数据缓存 Data、队列 Queue、浏览器功能测试 Support；最后详细分析了功能模块的源码实现，包括：属性操作 Attributes、事件系统 Events、DOM 遍历 Traversing、DOM 操作 Manipulation、样式操作 CSS、异步请求 Ajax、动画 Effects。

本书在分析每个模块时均采用由浅入深的方式，先概述功能、用法、结构和实现原理，然后介绍关键步骤和分析源码实现。让读者不仅知其然，而且知其所以然。事实上，本书的根本价值在于传达一种通过阅读源码快速成长的方式。无论是前端新人，还是经验丰富的老手，只要是对 JavaScript 感兴趣的开发人员，都会从本书中受益。



机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑：朱秀英

印刷

2014 年 1 月第 1 版第 1 次印刷

186mm × 240mm • 39.5 印张

标准书号：ISBN 978-7-111-44082-6

定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

前　　言

jQuery 是业界最流行的 JavaScript 库，其 API 非常精致和优雅，但是 jQuery 的源码却庞大且晦涩难懂，在本书开始写作时发布的 1.7.1 版本有 9266 行代码，涉及 17 个模块，读起来常常是一头雾水、有心无力。本书尝试对 jQuery 的源码进行系统、完整的介绍和分析，阐述 jQuery 的设计理念、实现原理和源码实现。

为什么要写这本书

笔者在 2010 年参与了一款卫星机顶盒用户界面的设计和开发，程序运行在机顶盒中间件供应商提供的一款定制浏览器上，在开发过程中，发现这款浏览器的行为类似于古老的 IE 5，各种缺陷和 bug 折磨得笔者苦不堪言，所以希望引入 jQuery 作为基础库，并开发一些通用组件和接口来简化开发过程，可是很快又发现这款浏览器对正则表达式的支持非常粗糙，导致选择器引擎 Sizzle 根本无法运行。此时，对 jQuery 进行简单改造已经满足不了需求。

然而令人惊艳的是，这款浏览器提供了与操作系统、文件系统、中间件、播放器、智能卡和卫星接收器等交互的 JavaScript API，例如，待机 & 关机、文件读写、计费、卫星锁频、数据接收等。鉴于这种复杂的体系架构，以及对浏览器缺陷的完善也非短期可以完成，笔者开始为这款机顶盒浏览器移植 jQuery，从而开始了对 jQuery 源码的学习和分析。

从 2011 年 6 月开始，笔者开始把心得和记录整理成《jQuery 1.6.1 源码分析系列》，陆续发表在程序员社区 [ITEye](#)^① 和 [博客园](#)^② 上，本书最初的内容也是基于这个系列而来的。《jQuery 1.6.1 源码分析系列》成体系但尚粗糙不堪，因此本书基于 jQuery 1.7.1 几乎全部重写，在内容上更加完善和严谨。

希望本书对读者能有所帮助。

读者对象

本书适合初级、中级、高级前端开发工程师，以及对前端开发感兴趣的读者。

在阅读本书之前，读者应该初步掌握 JavaScript、HTML、CSS 的基础知识，初步掌握 jQuery 的使用，或者有其他语言基础。

① <http://iteye.com>, <http://nuysoft.iteye.com/>。

② <http://cnblog.com>, <http://cnblogs.com/nuysoft/>。

如何阅读本书

本书共分为四大部分，首先介绍了 jQuery 的总体架构，然后分别分析了构造 jQuery 对象模块、底层支持模块和功能模块的源码实现。在阅读本书时，首先建议读者建立一个源码阅读和调试环境，在阅读过程中进行各种尝试和验证，加深对源码的理解；在阅读本书的每个章节前，建议读者先仔细阅读相应的官方文档，并验证官方示例，掌握 API 的功能和用法。

第一部分（第 1 章）对 jQuery 的设计理念、总体架构和源码结构进行了介绍和分析，让读者对 jQuery 有整体的认识。

第二部分（第 2 章）详细介绍和分析了构造函数 `jQuery()` 的用法、构造过程、原型属性和方法、静态属性和方法。

第三部分（第 3 ~ 7 章）详细分析了底层支持模块的源码实现，包括选择器 Sizzle、异步队列 Deferred Object、数据缓存 Data、队列 Queue、浏览器功能测试 Support。

第四部分（第 8 ~ 14 章）详细分析了功能模块的源码实现，包括属性操作 Attributes、事件系统 Events、DOM 遍历 Traversing、DOM 操作 Manipulation、样式操作 CSS、异步请求 Ajax、动画 Effects。

勘误和支持

由于笔者水平有限，再加上写作时的疏漏，书中难免存在许多需要改进之处。在此，欢迎读者朋友们指出书中存在的问题，并提出指导性意见，不胜感谢。提交地址为 <https://github.com/nuysoft/jquery-errata-support/issues>，勘误内容将在 <http://nuysoft.com/jquery.html> 上发布。

致谢

首先要向 jQuery 作者 John Resig、jQuery 团队和 jQuery 社区致敬，是你们繁荣了 JavaScript。

感谢机械工业出版社的杨绣国（Lisa）编辑，她对本书进行了大量润色，修订和批注的内容比我写的内容要多得多。本书的进度和内容在写作期间不断变更，因为她的不断激励和反复修订，才使本书得以呈现在广大读者面前。感谢朱秀英编辑，她修正了书稿中诸多不完善之处。感谢为本书拾遗补缺的诸多幕后编辑。

感谢为本书初稿给出反馈意见的所有人：古西风（叶克良）、左莫（徐波）、逸才（陈养剑）、崇志（李德强）、李志博、王阳光、符宝（徐丽丽）、余鹏、许杰。你们的宝贵意见使本书内容更加完善、更加准确。

本书最初的内容以及得到出版的机遇，源自发表在 ITeYe 和博客园的一系列博客，感谢这两个社区，以及因此结识的朋友们。

高云（nuysoft）

目 录

前言

第一部分 总体架构

第 1 章 总体架构	2
1.1 设计理念	2
1.2 总体架构	2
1.3 自调用匿名函数	4
1.4 总结	6

第二部分 构造 jQuery 对象

第 2 章 构造 jQuery 对象	8
2.1 构造函数 jQuery()	8
2.1.1 jQuery(selector [, context])	9
2.1.2 jQuery(html [, ownerDocument])、jQuery(html, props)	9
2.1.3 jQuery(element)、jQuery(elementArray)	10
2.1.4 jQuery(object)	10
2.1.5 jQuery(callback)	11
2.1.6 jQuery(jQuery object)	11
2.1.7 jQuery()	11
2.2 总体结构	11
2.3 jQuery.fn.init(selector, context, rootjQuery)	13
2.3.1 12 个分支	13
2.3.2 源码分析	14
2.3.3 小结	21
2.4 jQuery.buildFragment(args, nodes, scripts)	22
2.4.1 实现原理	22

2.4.2 源码分析	22
2.4.3 小结	26
2.5 jQuery.clean(elems, context, fragment, scripts)	27
2.5.1 实现原理	27
2.5.2 源码分析	27
2.5.3 小结	39
2.6 jQuery.extend()、jQuery.fn.extend()	40
2.6.1 如何使用	40
2.6.2 源码分析	40
2.7 原型属性和方法	43
2.7.1 .selector、.jquery、.length、.size()	44
2.7.2 .toArray()、.get([index])	45
2.7.3 .each(function(index, Element))、jQuery.each(collection, callback (indexInArray, valueOfElement))	46
2.7.4 .map(callback(index, domElement))、jQuery.map(arrayOrObject, callback(value, indexOrKey))	47
2.7.5 .pushStack(elements, name, arguments)	49
2.7.6 .end()	51
2.7.7 .eq(index)、.first()、.last()、.slice(start [, end])	51
2.7.8 .push(value, ...)、.sort([orderfunc])、.splice(start,deleteCount, value, ...)	52
2.7.9 小结	53
2.8 静态属性和方法	54
2.8.1 jQuery.noConflict([removeAll])	55
2.8.2 类型检测：jQueryisFunction(obj)、jQuery.isArray(obj)、jQuery.isWindow(obj)、jQuery.isNumeric(value)、jQuery.type(obj)、jQuery.isPlainObject(object)、jQuery.isEmptyObject(object)	56
2.8.3 解析 JSON 和 XML：jQuery.parseJSON(data)、jQuery.parseXML(data)	60
2.8.4 jQuery.globalEval(code)	65
2.8.5 jQuery.camelCase(string)	65
2.8.6 jQuery.nodeName(elem, name)	66
2.8.7 jQuery.trim(str)	67
2.8.8 数组操作方法：jQuery.makeArray(obj)、 jQuery.inArray(value, array [, fromIndex])、jQuery.merge(first, second)、 jQuery.grep(array, function(elementOfArray, indexInArray) [, invert])	68
2.8.9 jQuery.guid、jQuery.proxy(function, context)	72
2.8.10 jQuery.access(elems, key, value, exec, fn(elem, key, value), pass)	74
2.8.11 jQuery.error(message)、jQuery.noop()、jQuery.now()	75

2.8.12 浏览器嗅探：jQuery.uaMatch(ua)、jQuery.browser	76
2.8.13 小结	77
2.9 总结	77

第三部分 底层支持模块

第3章 选择器 Sizzle	80
3.1 总体结构	81
3.2 选择器表达式	83
3.3 设计思路	84
3.4 Sizzle(selector, context, results, seed)	86
3.5 正则 chunker	94
3.6 Sizzle.find(expr, context, isXML)	94
3.7 Sizzle.filter(expr, set, inplace, not)	99
3.8 Sizzle.selectors.relative	103
3.8.1 "+"	105
3.8.2 ">"	106
3.8.3 " "	108
3.8.4 "~"	108
3.8.5 dirCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)	109
3.8.6 dirNodeCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)	111
3.9 Sizzle.selectors	112
3.9.1 Sizzle.selectors.order	112
3.9.2 Sizzle.selectors.match/leftMatch	113
3.9.3 Sizzle.selectors.find	122
3.9.4 Sizzle.selectors.preFilter	123
3.9.5 Sizzle.selectors.filters	129
3.9.6 Sizzle.selectors.setFilters	132
3.9.7 Sizzle.selectors.filter	133
3.10 工具方法	140
3.10.1 Sizzle.uniqueSort(results)	140
3.10.2 sortOrder(a, b)	141
3.10.3 Sizzle.contains(a, b)	144
3.10.4 Sizzle.error(msg)	145
3.10.5 Sizzle.getText(elem)	145
3.11 便捷方法	146
3.11.1 Sizzle.matches(expr, set)	146

3.11.2 Sizzle.matchesSelector(node, expr)	146
3.12 jQuery 扩展	147
3.12.1 暴露 Sizzle 给 jQuery	147
3.12.2 .find(selector)	148
3.12.3 .has(target)	149
3.12.4 .not(selector)、.filter(selector)	150
3.12.5 .is(selector)	152
3.12.6 .closest(selectors, context)	153
3.12.7 .index(elem)	154
3.12.8 .add(selector, context)	155
3.12.9 jQuery.filter(expr, elems, not)	156
3.12.10 :animated	157
3.12.11 hidden、:visible	157
3.13 总结	158
第 4 章 异步队列 Deferred Object	160
4.1 jQuery.Callbacks(flags)	161
4.1.1 实现原理和总体结构	162
4.1.2 源码分析	163
4.1.3 小结	174
4.2 jQuery.Deferred(func)	174
4.2.1 实现原理和总体结构	176
4.2.2 源码分析	177
4.2.3 小结	183
4.3 jQuery.when(deferreds)	184
4.3.1 实现原理	185
4.3.2 源码分析	185
4.4 异步队列在 jQuery 中的应用	187
4.5 总结	188
第 5 章 数据缓存 Data	189
5.1 实现原理	189
5.1.1 为 DOM 元素附加数据	189
5.1.2 为 JavaScript 对象附加数据	191
5.2 总体结构	192
5.3 jQuery.acceptData(elem)	193
5.4 jQuery.data(elem, name, data, pvt)、jQuery._data(elem, name, data, pvt)	194

5.4.1 如何使用	194
5.4.2 源码分析	194
5.4.3 jQuery._data(elem, name, data)	199
5.4.4 小结	201
5.5 .data(key,value)	201
5.5.1 如何使用	201
5.5.2 源码分析	202
5.5.3 小结	206
5.6 jQuery.removeData(elem,name,pvt)、.removeData(key)	207
5.6.1 如何使用	207
5.6.2 源码分析	207
5.6.3 小结	212
5.7 .removeData(key)	213
5.8 jQuery.cleanData(elems)	213
5.8.1 应用场景	213
5.8.2 源码分析	214
5.8.3 小结	217
5.9 jQuery.hasData(elem)	217
5.10 总结	218
第 6 章 队列 Queue	219
6.1 如何使用	219
6.1.1 Ajax 队列	220
6.1.2 动画队列 + Ajax 队列	220
6.1.3 基于 JavaScript 对象	221
6.2 实现原理	221
6.3 总体结构	222
6.4 jQuery.queue(elem,type,data)	223
6.5 jQuery.dequeue(elem,type)	224
6.6 .queue(type,data)	227
6.7 .dequeue(type)	228
6.8 .delay(time,type)	229
6.9 .clearQueue(type)	230
6.10 jQuery._mark(elem,type)、jQuery._unmark(force,elem,type)	230
6.11 .promise(type,object)	232
6.11.1 如何使用	232
6.11.2 实现原理	233

6.11.3 源码分析	233
6.11.4 handleQueueMarkDefer(elem,type,src)	235
6.12 总结	237
第 7 章 浏览器功能测试 Support	238
7.1 总体结构	238
7.2 DOM 测试 (15 项).....	241
7.2.1 leadingWhitespace	241
7.2.2 tbody	242
7.2.3 htmlSerialize	243
7.2.4 hrefNormalized	245
7.2.5 checkOn	246
7.2.6 noCloneChecked	248
7.2.7 optSelected	250
7.2.8 optDisabled	251
7.2.9 getSetAttribute	253
7.2.10 deleteExpando	256
7.2.11 enctype	258
7.2.12 html5Clone	259
7.2.13 radioValue	261
7.2.14 checkClone	263
7.2.15 appendChecked	264
7.3 样式测试 (3 项).....	266
7.3.1 style	266
7.3.2 opacity	268
7.3.3 cssFloat	272
7.4 盒模型测试 (10 项).....	273
7.4.1 reliableMarginRight	273
7.4.2 reliableHiddenOffsets	276
7.4.3 boxModel	278
7.4.4 inlineBlockNeedsLayout	280
7.4.5 shrinkWrapBlocks	282
7.4.6 doesNotAddBorder、doesAddBorderForTableAndCells	285
7.4.7 fixedPosition	287
7.4.8 subtractsBorderForOverflowNotVisible	290
7.4.9 doesNotIncludeMarginInBodyOffset	292
7.5 事件测试 (4 项).....	294

7.5.1	noCloneEvent	294
7.5.2	submitBubbles、changeBubbles、focusinBubbles	296
7.6	Ajax 测试（2项）.....	298
7.6.1	ajax	298
7.6.2	cors	300
7.7	总结	301

第四部分 功能模块

第 8 章 属性操作 Attributes	306
8.1 总体结构	307
8.2 jQuery.attr(elem, name, value, pass)	308
8.2.1 源码分析	308
8.2.2 boolHook	311
8.2.3 nodeHook	313
8.2.4 jQuery.attrHooks	314
8.2.5 小结	319
8.3 .attr(name, value)	319
8.4 jQueryremoveAttr(elem, value)	321
8.4.1 源码分析	321
8.4.2 小结	322
8.5 .removeAttr(name)	323
8.6 jQuery.prop(elem, name, value)	323
8.6.1 源码分析	323
8.6.2 jQuery.propHooks	325
8.6.3 小结	326
8.7 .prop(name, value)	327
8.8 .removeProp(name)	327
8.9 .addClass(className)	328
8.9.1 源码分析	328
8.9.2 小结	330
8.10 .removeClass([className])	330
8.10.1 源码分析	331
8.10.2 小结	333
8.11 .toggleClass([className][, switch])	333
8.11.1 源码分析	334
8.11.2 小结	336

8.12 .hasClass(selector)	336
8.12.1 源码分析	336
8.12.2 小结	337
8.13 .val([value])	338
8.13.1 源码分析	338
8.13.2 jQuery.valHooks	340
8.13.3 小结	343
8.14 总结	344
第 9 章 事件系统 Events	346
9.1 总体结构	346
9.2 实现原理	350
9.3 jQuery 事件对象	353
9.3.1 构造函数 jQuery.Event(src, props)	355
9.3.2 原型对象 jQuery.Event.prototype	357
9.3.3 事件属性修正方法 jQuery.event.fix(event)	360
9.4 绑定事件	367
9.4.1 .on(events [, selector] [, data] , handler(eventObject))	367
9.4.2 jQuery.event.add(elem, types, handler, data, selector)	370
9.5 移除事件	379
9.5.1 .off(events [, selector] [, handler(eventObject)])	379
9.5.2 jQuery.event.remove(elem, types, handler, selector, mappedTypes)	382
9.6 事件响应	388
9.6.1 主监听函数	388
9.6.2 jQuery.event.dispatch(event)	390
9.7 手动触发事件	396
9.7.1 .trigger(eventType [, extraParameters])、.triggerHandler(eventType [, extraParameters])	396
9.7.2 jQuery.event.trigger(event, data, elem, onlyHandlers)	397
9.8 事件修正和模拟 jQuery.event.special	406
9.8.1 ready	408
9.8.2 load	408
9.8.3 focus、blur	409
9.8.4 beforeunload	409
9.8.5 mouseenter、mouseleave	410
9.8.6 submit	412
9.8.7 change	413

9.8.8 focusin、focusout	416
9.8.9 jQuery.event.simulate(type, elem, event, bubble)	417
9.9 事件便捷方法	418
9.10 组合方法	419
9.10.1 .toggle(handler(eventObject), handler(eventObject) [, handler(eventObject)])	419
9.10.2 .hover(handlerIn(eventObject) [, handlerOut(eventObject)])	421
9.11 ready 事件	421
9.11.1 总体结构	421
9.11.2 .ready(handler)	424
9.11.3 jQuery.bindReady()	424
9.11.4 jQuery.holdReady(hold)	427
9.11.5 jQuery.ready(wait)	428
9.12 总结	430
第 10 章 DOM 遍历 Traversing	433
10.1 总体结构	434
10.2 遍历函数	435
10.3 工具函数	437
10.3.1 jQuery.dir(elem, dir, until)	437
10.3.2 jQuery.nth(cur, result, dir, elem)	439
10.3.3 jQuery.sibling(n, elem)	440
10.4 模板函数	441
10.5 总结	443
第 11 章 DOM 操作 Manipulation	444
11.1 总体结构	444
11.2 插入元素	445
11.2.1 核心方法 .domManip(args, table, callback)	445
11.2.2 .append(content [, content])	451
11.2.3 .prepend(content [, content])	452
11.2.4 .before(content [, content])	452
11.2.5 .after(content [, content])	452
11.2.6 .appendTo(target)、.prependTo(target)、.insertBefore(target)、.insertAfter(target)	453
11.2.7 .html([value])	454
11.2.8 .text([text])	458
11.3 删除元素	459

11.3.1 .remove(selector, keepData)	459
11.3.2 .empty()	460
11.3.3 .detach(selector)	460
11.4 复制元素	461
11.4.1 .clone(dataAndEvents, deepDataAndEvents)	461
11.4.2 jQuery.clone(elem, dataAndEvents, deepDataAndEvents)	461
11.4.3 cloneFixAttributes(src, dest)	465
11.5 替换元素	467
11.5.1 .replaceWith(value)	467
11.5.2 .replaceAll(target)	469
11.6 包裹元素	469
11.6.1 .wrapAll(html)	469
11.6.2 .wrapInner(html)	470
11.6.3 .wrap(html)	471
11.6.4 .unwrap()	471
11.7 总结	472
第 12 章 样式操作 CSS	474
12.1 内联样式、计算样式	475
12.1.1 总体结构	475
12.1.2 .css(name, value)	476
12.1.3 jQuery.style(elem, name, value, extra)	477
12.1.4 jQuery.css(elem, name, extra)	481
12.1.5 curCSS(elem, name)、getComputedStyle(elem, name)、 currentStyle(elem, name)	483
12.1.6 jQuery.cssHooks	486
12.2 坐标 Offset	492
12.2.1 总体结构	492
12.2.2 .offset(options)	493
12.2.3 jQuery.offset.setOffset(elem, options, i)	498
12.2.4 jQuery.offset.bodyOffset(body)	500
12.2.5 .position()	501
12.2.6 .offsetParent()	502
12.2.7 .scrollLeft(val)、.scrollTop(val)	503
12.3 尺寸 Dimensions	504
12.3.1 总体结构	504
12.3.2 getWH(elem, name, extra)	505

12.3.3 .innerHeight()、.innerWidth()	508
12.3.4 .outerHeight(margin)、.outerWidth(margin)	509
12.3.5 .height(size)、.width(size)	509
12.3.6 小结	513
12.4 总结	513
第 13 章 异步请求 Ajax	516
13.1 总体结构	517
13.2 jQuery.ajax(url, options)	519
13.3 前置过滤器、请求发送器的初始化和执行	540
13.3.1 初始化	540
13.3.2 执行	543
13.4 前置过滤器	545
13.4.1 json、jsonp	545
13.4.2 script	548
13.4.3 小结	549
13.5 请求发送器	549
13.5.1 script	549
13.5.2 XMLHttpRequest	552
13.5.3 小结	560
13.6 数据转换器	561
13.6.1 初始化	561
13.6.2 执行	562
13.6.3 小结	566
13.7 Ajax 事件	567
13.8 便捷方法	568
13.8.1 jQuery.get(url, data, callback, type)、jQuery.post(url, data, callback, type)	569
13.8.2 jQuery.getJSON(url, data, callback)、jQuery.getScript(url, callback)	569
13.8.3 .load(url, params, callback)	570
13.9 工具方法	573
13.9.1 .serialize()	573
13.9.2 jQuery.param(a, traditional)	574
13.9.3 .serializeArray()	577
13.10 总结	579
第 14 章 动画 Effects	582
14.1 总体结构	583

14.2 动画入口	586
14.2.1 .animate(prop, speed, easing, callback)	586
14.2.2 jQuery.speed(speed, easing, fn)	588
14.2.3 doAnimation()	590
14.2.4 jQuery.fx(elem, options, prop)	595
14.2.5 jQuery.fx.prototype.show()	595
14.2.6 jQuery.fx.prototype.hide()	596
14.2.7 小结	596
14.3 动画执行	597
14.3.1 jQuery.fx.prototype.custom(from, to, unit)	598
14.3.2 jQuery.fx.tick()	599
14.3.3 jQuery.fx.prototype.step(gotoEnd)	600
14.3.4 jQuery.easing	604
14.3.5 jQuery.fx.prototype.update()	604
14.3.6 jQuery.fx.step	605
14.4 停止动画 .stop(type, clearQueue, gotoEnd)	606
14.5 便捷方法	609
14.5.1 生成动画样式集 genFx(type, num)	609
14.5.2 显示隐藏 .show/hide/toggle()	610
14.5.3 渐显渐隐 .fadeIn/fadeOut/fadeTo/fadeToggle()	613
14.5.4 滑入滑出 .slideDown/slidUp/slideToggle()	614
14.6 总结	615

华章图书

第一部分

总体架构

□ 第1章 总体架构



第 1 章

总体架构

1.1 设计理念

jQuery 是一款革命性的 JavaScript 库，秉承着“以用为本”的设计理念，倡导“写更少的代码，做更多的事”(write less, do more)，极大地提升了 JavaScript 开发体验。

jQuery 的核心特性可以总结为：

- 兼容主流浏览器^①，支持 IE 6.0+、Chrome、Firefox 3.6+、Safari 5.0+、Opera 等。
- 具有独特的链式语法和短小清晰的多功能接口。
- 具有高效灵活的 CSS 选择器，并且可对 CSS 选择器进行扩展。
- 拥有便捷的插件扩展机制和丰富的插件。

1.2 总体架构

jQuery 的模块可以分为 3 部分：入口模块、底层支持模块和功能模块，如图 1-1 所示，图中还展示了模块之间的主要依赖关系。

来看看图 1-1 中各个模块的功能和依赖关系。

在构造 jQuery 对象模块中，如果在调用构造函数 `jQuery()` 创建 jQuery 对象时传入了选择器表达式，则会调用选择器 Sizzle 遍历文档，查找与之匹配的 DOM 元素，并创建一个包含了这些 DOM 元素引用的 jQuery 对象。

选择器 Sizzle 是一款纯 JavaScript 实现的 CSS 选择器引擎，用于查找与选择器表达式匹配的元素集合。

工具方法模块提供了一些编程辅助方法，用于简化对 jQuery 对象、DOM 元素、数组、对象、字符串等的操作，例如，`jQuery.each()`、`.each()`、`jQuery.map()`、`.map()` 等，其他所有的模块都会用到工具方法模块。

^① 本书基于 jQuery 1.7.1 编写。在本书写作期间发布的 jQuery 2.x 不再支持 IE 9.0 以下的浏览器，请参见 <http://jquery.com/browser-support/>。

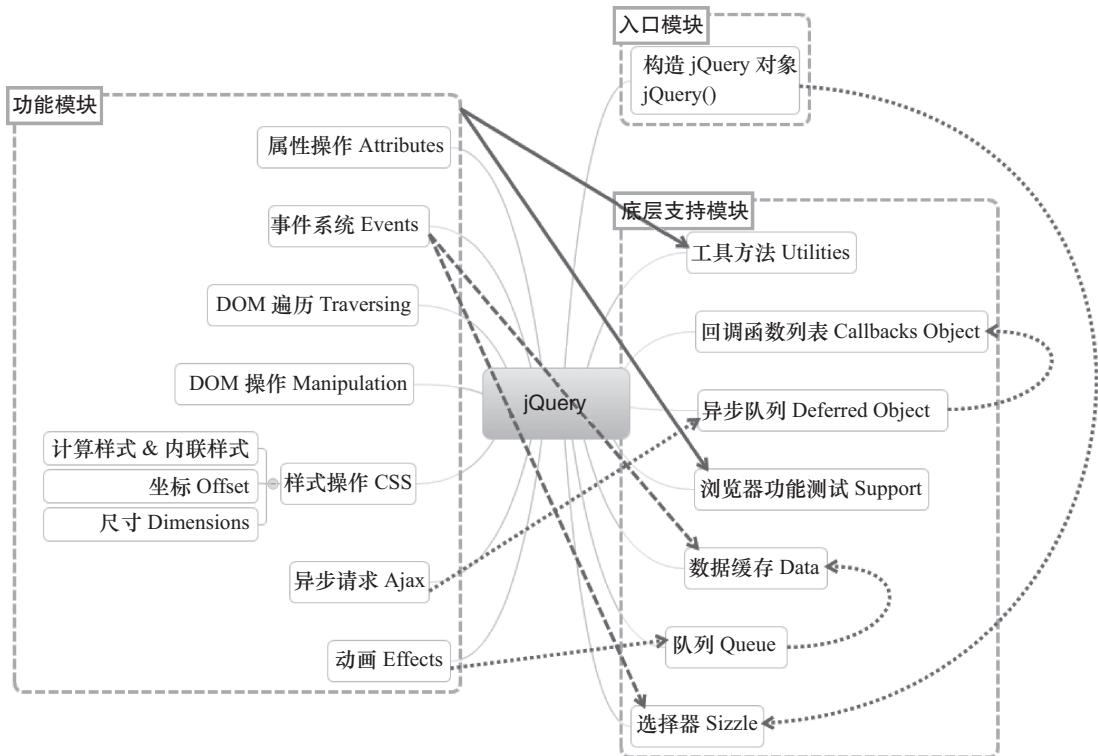


图 1-1 jQuery 的模块分类和主要依赖关系

浏览器功能测试模块提供了针对不同浏览器功能和 bug 的测试结果，其他模块则基于这些测试结果来解决浏览器之间的兼容性问题。

在底层支持模块中，回调函数列表模块用于增强对回调函数的管理，支持添加、移除、触发、锁定、禁用回调函数等功能；异步队列模块用于解耦异步任务和回调函数，它在回调函数列表的基础上为回调函数增加了状态，并提供了多个回调函数列表，支持传播任意同步或异步回调函数的成功或失败状态；数据缓存模块用于为 DOM 元素和 JavaScript 对象附加任意类型的数据；队列模块用于管理一组函数，支持函数的入队和出队操作，并确保函数按顺序执行，它基于数据缓存模块实现。

在功能模块中，事件系统提供了统一的事件绑定、响应、手动触发和移除机制，它并没有将事件直接绑定到 DOM 元素上，而是基于数据缓存模块来管理事件；Ajax 模块允许从服务器上加载数据，而不用刷新页面，它基于异步队列模块来管理和触发回调函数；动画模块用于向网页中添加动画效果，它基于队列模块来管理和执行动画函数；属性操作模块用于对 HTML 属性和 DOM 属性进行读取、设置和移除操作；DOM 遍历模块用于在 DOM 树中遍历父元素、子元素和兄弟元素；DOM 操作模块用于插入、移除、复制和替换 DOM 元素；样式操作模块用于获取计算样式或设置内联样式；坐标模块用于读取或设置 DOM 元素的文档坐标；尺寸模块用于获取 DOM 元素的高度和宽度。

下面来看看jQuery源码(jquery-1.7.1.js)的总体结构，如代码清单1-1所示，其中展示了各个模块在源码中的位置。

代码清单1-1 jQuery源码(jquery-1.7.1.js)的总体结构

```
(function( window, undefined ) {
    // 构造jQuery对象
    var jQuery = (function() {
        var jQuery = function( selector, context ) {
            return new jQuery.fn.init( selector, context, rootjQuery );
        }
        return jQuery;
    })();
    // 工具方法 Utilities
    // 回调函数列表 Callbacks Object
    // 异步队列 Deferred Object
    // 浏览器功能测试 Support
    // 数据缓存 Data
    // 队列 Queue
    // 属性操作 Attributes
    // 事件系统 Events
    // 选择器 Sizzle
    // DOM 遍历 Traversing
    // DOM 操作 Manipulation
    // 样式操作 CSS (计算样式、内联样式)
    // 异步请求 Ajax
    // 动画 Effects
    // 坐标 Offset、尺寸 Dimensions
    window.jQuery = window.$ = jQuery;
}) (window);
```

从代码清单1-1可以看出，jQuery的源码结构还是相当清晰和有条理的，并不像源码那般晦涩。

1.3 自调用匿名函数

从代码清单1-1中还可以看到，jQuery的所有代码都被包裹在了一个立即执行的匿名函数表达式中，这种代码结构称为“自调用匿名函数”。当浏览器加载完jQuery文件后，自调用匿名函数会立即开始执行，初始化jQuery的各个模块。相关代码如下所示：

```
(function( window, undefined ) {
    var jQuery = ...
    // ...
    window.jQuery = window.$ = jQuery;
}) (window);
```

上面这段代码涉及一些JavaScript基础知识和编码习惯，下面以提问的方式来逐一分析。

1) 为什么要创建这样一个自调用匿名函数？

通过创建一个自调用匿名函数，创建了一个特殊的函数作用域，该作用域中的代码不会和已有的同名函数、方法和变量以及第三方库冲突。由于jQuery会被应用在成千上万的JavaScript程序中，所以必须确保jQuery的代码不会受到其他代码的干扰，并且jQuery不能

破坏和污染全局变量以至于影响到其他代码。这一点是任何一个 JavaScript 库和框架所必须具备的功能。

注意，在这个自调用匿名函数的最后，通过手动把变量 jQuery 添加到 window 对象上，明确地使变量 jQuery 成为公开的全局变量，而其他的部分将是私有的。

另外，自调用匿名函数还可以有两种等价的写法，如下所示（注意加了底纹的圆括号的位置）：

```
// 写法 1 (常见写法，也是 jQuery 所采用的)
( function() {
    // ...
} )();
```



```
// 写法 2
( function() {
    // ...
} () );
```



```
// 写法 3
!function() {
    // ...
} ();
```

2) 为什么要为自调用匿名函数设置参数 window，并传入 window 对象？

通过传入 window 对象，可以使 window 对象变为局部变量（即把函数参数作为局部变量使用），这样当在 jQuery 代码块中访问 window 对象时，不需要将作用域链回退到顶层作用域，从而可以更快地访问 window 对象，这是原因之一；另外，将 window 对象作为参数传入，可以在压缩代码时进行优化，在压缩文件 jquery-1.7.1.min.js 中可以看到下面的代码：

```
(function(a,b){ ... })(window);
// 参数 window 被压缩为 a，参数 undefined 被压缩为 b
```

3) 为什么要为自调用匿名函数设置参数 undefined？

特殊值 undefined 是 window 对象的一个属性，例如，执行下面的代码将会弹出 true：

```
alert( "undefined" in window ); // true
```

通过把参数 undefined 作为局部变量使用，但是又不传入任何值，可以缩短查找 undefined 时的作用域链，并且可以在压缩代码时进行优化，如前面代码所示，参数 undefined 会被压缩为 b。

另外，更重要的原因是，通过这种方式可以确保参数 undefined 的值是 undefined，因为 undefined 有可能会被重写为新的值。可以用下面的代码来尝试修改 undefined 的值，在各浏览器中的测试结果见表 1-1。

```
undefined = "now it's defined";
alert( undefined );
```

表 1-1 在浏览器中尝试修改 undefined 的值

浏览器	测试结果	结论
IE 6.0、IE 7.0、IE 8.0	now it's defined	可以改变
IE 9.0、IE 10.0	undefined	不能改变
Chrome 16.0.912.77	now it's defined	可以改变
Chrome 17.0.963.56	undefined	不能改变
Firefox 3.6.28	now it's defined	可以改变
Firefox 4.0	undefined	不能改变
Safari 4.0.2	now it's defined	可以改变
Safari 4.0.4	undefined	不能改变
Opera 11.52	now it's defined	可以改变
Opera 11.60	undefined	不能改变

4) 注意到自调用匿名函数最后的分号(;)了吗?

通常在 JavaScript 中, 如果语句分别放置在不同的行中, 则分号(;)是可选的, 但是对于自调用匿名函数来说, 在之前或之后省略分号都可能会引起语法错误。例如, 执行下面的两个例子, 就会抛出异常。

例 1 在下面的代码中, 如果自调用匿名函数的前一行末尾没有加分号, 则自调用匿名函数的第一对括号会被当作是函数调用。

```
var n = 1
( function(){} )()
//TypeError: number is not a function
```

例 2 在下面的代码中, 如果未在第一个自调用匿名函数的末尾加分号, 则下一行自调用匿名函数的第一对括号会被当作是函数调用。

```
( function(){} )()
( function(){} )()
//TypeError: undefined is not a function
```

所以, 在使用自调用匿名函数时, 最好不要省略自调用匿名函数之前和之后的分号。

1.4 总结

在本章的前半部分, 对 jQuery 的总体架构进行了梳理, 并对各个模块的分类、功能和主要依赖关系做了简要介绍。通过这些介绍, 读者已经对 jQuery 的代码有了整体上的认识。

在后半部分, 则以提问的方式对包裹 jQuery 代码的自调用匿名函数进行了分析, 扫除了读者阅读 jQuery 源码的第一道障碍。

第二部分

构造 jQuery 对象

□ 第 2 章 构造 jQuery 对象



第 2 章

构造 jQuery 对象

jQuery 对象是一个类数组对象，含有连续的整型属性、length 属性和大量的 jQuery 方法。jQuery 对象由构造函数 `jQuery()` 创建，`$()` 则是 `jQuery()` 的缩写。

2.1 构造函数 `jQuery()`

如果调用构造函数 `jQuery()` 时传入的参数不同，创建 jQuery 对象的逻辑也会随之不同。构造函数 `jQuery()` 有 7 种用法，如图 2-1 所示。



图 2-1 构造函数 `jQuery()`

下面分别介绍构造函数 `jQuery()` 的 7 种用法。

2.1.1 `jQuery(selector [, context])`

如果传入一个字符串参数，jQuery 会检查这个字符串是选择器表达式还是 HTML 代码。如果是选择器表达式，则遍历文档，查找与之匹配的 DOM 元素，并创建一个包含了这些 DOM 元素引用的 jQuery 对象；如果没有元素与之匹配，则创建一个空 jQuery 对象，其中不包含任何元素，其属性 `length` 等于 0。字符串参数是 HTML 代码的情况会在下一小节介绍。

默认情况下，对匹配元素的查找将从根元素 `document` 对象开始，即查找范围是整个文档树，不过也可以传入第二个参数 `context` 来限定查找范围（本书中把参数 `context` 称为“选择器的上下文”，或简称“上下文”）。例如，在一个事件监听函数中，可以像下面这样限制查找范围：

```
$('div.foo').click(function() {
    $('span', this).addClass('bar'); // 限定查找范围
});
```

在这个例子中，对选择器表达式“`span`”的查找被限制在了 `this` 的范围内，即只有被点击元素内的 `span` 元素才会被添加类样式“`bar`”。

如果选择器表达式 `selector` 是简单的“`#id`”，且没有指定上下文 `context`，则调用浏览器原生方法 `document.getElementById()` 查找属性 `id` 等于指定值的元素；如果是比“`#id`”复杂的选择器表达式或指定了上下文，则通过 jQuery 方法 `.find()` 查找，因此 `$('span', this)` 等价于 `$(this).find('span')`。

至于方法 `.find()`，会调用 CSS 选择器引擎 Sizzle 实现，在第 3 章中将会进行介绍和分析。

2.1.2 `jQuery(html [, ownerDocument])`、`jQuery(html, props)`

如果传入的字符串参数看起来像一段 HTML 代码（例如，字符串中含有 `<tag...>`），jQuery 则尝试用这段 HTML 代码创建新的 DOM 元素，并创建一个包含了这些 DOM 元素引用的 jQuery 对象。例如，下面的代码将把 HTML 代码转换成 DOM 元素并插入 `body` 节点的末尾：

```
$('<p id="test">My <em>new</em> text</p>').appendTo('body');
```

如果 HTML 代码是一个单独标签，例如，`$('')` 或 `$('<a>')`，jQuery 会使用浏览器原生方法 `document.createElement()` 创建 DOM 元素。如果是比单独标签更复杂的 HTML 片段，例如上面例子中的 `$('<p id = "test">Mynewtext</p>')`，则利用浏览器的 `innerHTML` 机制创建 DOM 元素，这个过程由方法 `jQuery.buildFragment()` 和方法 `jQuery.clean()` 实现，相关内容分别在 2.4 节和 2.5 节介绍和分析。

第二个参数 `ownerDocument` 用于指定创建新 DOM 元素的文档对象，如果不传入，则默认为当前文档对象。

如果 HTML 代码是一个单独标签，那么第二个参数还可以是 `props`，`props` 是一个包含了属性、事件的普通对象；在调用 `document.createElement()` 创建 DOM 元素后，参数 `props` 会

被传给jQuery方法`.attr()`，然后由`.attr()`负责把参数`props`中的属性、事件设置到新创建的DOM元素上。

参数`props`的属性可以是任意的事件类型（如“`click`”），此时属性值应该是事件监听函数，它将被绑定到新创建的DOM元素上；参数`props`可以含有以下特殊属性：`val`、`css`、`html`、`text`、`data`、`width`、`height`、`offset`，相应的jQuery方法：`.val()`、`.css()`、`.html()`、`.text()`、`.data()`、`.width()`、`.height()`、`.offset()`将被执行，并且属性值会作为参数传入；其他类型的属性则会被设置到新创建的DOM元素上，某些特殊属性还会做跨浏览器兼容（如`type`、`value`、`tabindex`等）；可以通过属性名`class`设置类样式，但要用引号把`class`包裹起来，因为`class`是JavaScript保留字。例如，在下面的例子中，创建一个`div`元素，并设置类样式为“`test`”、设置文本内容为“`Click me!`”、绑定一个`click`事件，然后插入`body`节点的末尾，当点击该`div`元素时，还会切换类样式`test`：

```
$("<div/>", {
  "class": "test",
  text: "Click me!",
  click: function() {
    $(this).toggleClass("test");
  }
}).appendTo("body");
```

方法`.attr()`将在8.2节介绍和分析。

2.1.3 jQuery(element)、jQuery(elementArray)

如果传入一个DOM元素或DOM元素数组，则把DOM元素封装到jQuery对象中并返回。

这个功能常见于事件监听函数，即把关键字`this`引用的DOM元素封装为jQuery对象，然后在该jQuery对象上调用jQuery方法。例如，在下面的例子中，先调用`$(this)`把被点击的`div`元素封装为jQuery对象，然后调用方法`slideUp()`以滑动动画隐藏该`div`元素：

```
$('div.foo').click(function() {
  $(this).slideUp();
});
```

2.1.4 jQuery(object)

如果传入一个普通JavaScript对象，则把该对象封装到jQuery对象中并返回。

这个功能可以方便地在普通JavaScript对象上实现自定义事件的绑定和触发，例如，执行下面的代码会在对象`foo`上绑定一个自定义事件`custom`，然后手动触发这个事件，执行绑定的`custom`事件监听函数，如下所示：

```
// 定义一个普通 JavaScript 对象
var foo = {foo:'bar', hello:'world'};
// 封装成 jQuery 对象
var $foo = $(foo);
// 绑定一个事件
```

```

$foo.on('custom', function () {
    console.log('custom event was called');
});
// 触发这个事件
$foo.trigger('custom');                                // 在控制台打印 "custom event was called"

```

2.1.5 jQuery(callback)

如果传入一个函数，则在 document 上绑定一个 ready 事件监听函数，当 DOM 结构加载完成时执行。ready 事件的触发要早于 load 事件。ready 事件并不是浏览器原生事件，而是 DOMContentLoaded 事件、onreadystatechange 事件和函数 doScrollCheck() 的统称，将在 9.11 节介绍和分析。

2.1.6 jQuery(jQuery object)

如果传入一个 jQuery 对象，则创建该 jQuery 对象的一个副本并返回，副本与传入的 jQuery 对象引用完全相同的 DOM 元素。

2.1.7 jQuery()

如果不传入任何参数，则返回一个空的 jQuery 对象，属性 length 为 0。注意，在 jQuery 1.4 之前，会返回一个含有 document 对象的 jQuery 对象。

这个功能可以用来复用 jQuery 对象，例如，创建一个空的 jQuery 对象，然后在需要时先手动修改其中的元素，再调用 jQuery 方法，从而避免重复创建 jQuery 对象。

2.2 总体结构

构造 jQuery 对象模块的总体源码结构如代码清单 2-1 所示。

代码清单 2-1 构造 jQuery 对象模块的总体源码结构

```

16  (function( window, undefined ) {
17      // 构造 jQuery 对象
18      var jQuery = (function() {
19          var jQuery = function( selector, context ) {
20              return new jQuery.fn.init( selector, context, rootjQuery );
21          },
22              // 一堆局部变量声明
23              jQuery.fn = jQuery.prototype = {
24                  constructor: jQuery,
25                  init: function( selector, context, rootjQuery ) { ... },
26                  // 一堆原型属性和方法
27              };
28          jQuery.fn.init.prototype = jQuery.fn;
29          jQuery.extend = jQuery.fn.extend = function() { ... };
30          jQuery.extend({
31              // 一堆静态属性和方法
32          });
33      });
34      return jQuery;
35  }

```

```

957     })();
// 省略其他模块的代码
9246     window.jQuery = window.$ = jQuery;
9266 })( window );

```

下面简要梳理下这段源码。

第16 ~ 9266行是最外层的自调用匿名函数，第1章中介绍过，当jQuery初始化时，这个自调用匿名函数包含的所有JavaScript代码将被执行。

第22行定义了一个变量jQuery，第22 ~ 957行的自调用匿名函数返回jQuery构造函数并赋值给变量jQuery，最后在第9246行把这个jQuery变量暴露给全局作用域window，并定义了别名\$。

在第22 ~ 957行的自调用匿名函数内，第25行又定义了一个变量jQuery，它的值是jQuery构造函数，在第955行返回并赋值给第22行的变量jQuery。因此，这两个jQuery变量是等价的，都指向jQuery构造函数，为了方便描述，在后文中统一称为构造函数jQuery()。

第97 ~ 319行覆盖了构造函数jQuery()的原型对象。第98行覆盖了原型对象的属性constructor，使它指向jQuery构造函数；第99行定义了原型方法jQuery.fn.init()，它负责解析参数selector和context的类型并执行相应的查找；在第27行可以看到，当我们调用jQuery构造函数时，实际返回的是jQuery.fn.init()的实例；此外，还定义了一堆其他的原型属性和方法，例如，selector、length、size()、toArray()等。

第322行用jQuery构造函数的原型对象jQuery.fn覆盖了jQuery.fn.init()的原型对象。

第324行定义了jQuery.extend()和jQuery.fn.extend()，用于合并两个或多个对象的属性到第一个对象；第388 ~ 892行执行jQuery.extend()在jQuery构造函数上定义了一堆静态属性和方法，例如，noConflict()、isReady、readyWait、holdReady()等。

看上去代码清单2-1所述的总体源码结构有些复杂，下面把疑问和难点一一罗列，逐个分析。

1) 为什么要在构造函数jQuery()内部用运算符new创建并返回另一个构造函数的实例？

通常我们创建一个对象或实例的方式是在运算符new后紧跟一个构造函数，例如，newDate()会返回一个Date对象；但是，如果构造函数有返回值，运算符new所创建的对象会被丢弃，返回值将作为new表达式的值。

jQuery利用了这一特性，通过在构造函数jQuery()内部用运算符new创建并返回另一个构造函数的实例，省去了构造函数jQuery()前面的运算符new，即我们创建jQuery对象时，可以省略运算符new直接写jQuery()。

为了拼写更方便，在第9246行还为构造函数jQuery()定义了别名\$，因此，创建jQuery对象的常见写法是\$()。

2) 为什么在第97行执行jQuery.fn = jQuery.prototype，设置jQuery.fn指向构造函数jQuery()的原型对象jQuery.prototype？

jQuery.fn是jQuery.prototype的简写，可以少写7个字符，以方便拼写。

3) 既然调用构造函数jQuery()返回的jQuery对象实际上是构造函数jQuery.fn.init()的实例，为什么能在构造函数jQuery.fn.init()的实例上调用构造函数jQuery()的原型方法和属性？例如，\$('#id').length和\$('#id').size()。

在第 322 行执行 `jQuery.fn.init.prototype = jQuery.fn` 时，用构造函数 `jQuery()` 的原型对象覆盖了构造函数 `jQuery.fn.init()` 的原型对象，从而使构造函数 `jQuery.fn.init()` 的实例也可以访问构造函数 `jQuery()` 的原型方法和属性。

4) 为什么要把第 25 ~ 955 行的代码包裹在一个自调用匿名函数中，然后把第 25 行定义的构造函数 `jQuery()` 作为返回值赋值给第 22 行的 `jQuery` 变量？去掉这个自调用匿名函数，直接在第 25 行定义构造函数 `jQuery()` 不也可以吗？去掉了不是更容易阅读和理解吗？

去掉第 25 ~ 955 行的自调用匿名函数当然可以，但会潜在地增加构造 `jQuery` 对象模块与其他模块的耦合度。在第 25 ~ 97 行之间还定义了很多其他的局部变量，这些局部变量只在构造 `jQuery` 对象模块内部使用。通过把这些局部变量包裹在一个自调用匿名函数中，实现了高内聚低耦合的设计思想。

5) 为什么要覆盖构造函数 `jQuery()` 的原型对象 `jQuery.prototype` ?

在原型对象 `jQuery.prototype` 上定义的属性和方法会被所有 `jQuery` 对象继承，可以有效减少每个 `jQuery` 对象所需的内存。事实上，`jQuery` 对象只包含 5 种非继承属性，其余都继承自原型对象 `jQuery.prototype`；在构造函数 `jQuery.fn.init()` 中设置了整型属性、`length`、`selector`、`context`；在原型方法 `.pushStack()` 中设置了 `prevObject`。因此，也不必因为 `jQuery` 对象带有太多的属性和方法而担心会占用太多的内存。

关于构造函数、原型、继承等基础知识，请查阅相关的基础类书籍。

2.3 jQuery.fn.init(selector, context, rootjQuery)

2.3.1 12 个分支

构造函数 `jQuery.fn.init()` 负责解析参数 `selector` 和 `context` 的类型，并执行相应的逻辑，最后返回 `jQuery.fn.init()` 的实例。参数 `selector` 和 `context` 共有 12 个有效分支，如表 2-1 所示。

表 2-1 参数 selector 和 context 的 12 个分支

	selector	context	示例
1	可以转换为 false	—	<code>\$()</code>
2	DOM 元素	—	<code>\$(document.body)</code>
3	字符串	“body”	<code>\$('body')</code>
4		单独标签	<code>\$('<div>')</code> <code>\$('<div>', { 'class': 'test' })</code>
5		复杂 HTML 代码	<code>\$('<div>abc</div>')</code>
6		“#id”	<code>\$('#id')</code>
7		选择器表达式	<code>\$('div p')</code>
8		选择器表达式	<code>\$('div p', \$('#id'))</code>
9		选择器表达式	<code>\$('div.foo').click(function() {</code> <code>\$('span', this).addClass('bar');</code> <code>});</code>

(续)

	selector	context	示例
10	函数	—	<code>\$(function(){ ... })</code>
11	jQuery 对象	—	<code>\$(\$('div p'))</code>
12	其他任意类型的值	—	<code>\$({ abc: 123 })</code> <code>\$([1, 2, 3])</code>

下面分析 `jQuery.fn.init()` 的源码，看看它是如何解析和处理参数 `selector` 和 `context` 的 12 个分支的。

2.3.2 源码分析

1. 定义 `jQuery.fn.init(selector, context, rootjQuery)`

相关代码如下所示：

```
99     init: function( selector, context, rootjQuery ) {
100         var match, elem, ret, doc;
```

第 99 行：定义构造函数 `jQuery.fn.init(selector, context, rootjQuery)`，它接受 3 个参数：

- 参数 `selector`：可以是任意类型的值，但只有 `undefined`、DOM 元素、字符串、函数、`jQuery` 对象、普通 JavaScript 对象这几种类型是有效的，其他类型的值也可以接受但没有意义。
- 参数 `context`：可以不传入，或者传入 DOM 元素、`jQuery` 对象、普通 JavaScript 对象之一。
- 参数 `rootjQuery`：包含了 `document` 对象的 `jQuery` 对象，用于 `document.getElementById()` 查找失败、`selector` 是选择器表达式且未指定 `context`、`selector` 是函数的情况。`rootjQuery` 的定义和应用场景的代码如下所示：

```
// document.getElementById() 查找失败
172             return rootjQuery.find( selector );
// selector 是选择器表达式且未指定 context
187             return ( context || rootjQuery ).find( selector );
// selector 是函数
198             return rootjQuery.ready( selector );

// 定义 rootjQuery
916 // All jQuery objects should point back to these
917 rootjQuery = jQuery(document);
918
```

第 100 行：变量 `match`、`elem`、`ret`、`doc` 的功能会在接下来的分析过程中介绍。

2. 参数 `selector` 可以转换为 `false`

参数 `selector` 可以转换为 `false`，例如是 `undefined`、空字符串、`null` 等，则直接返回 `this`，此时 `this` 是空 `jQuery` 对象，其属性 `length` 等于 0。相关代码如下所示：

```

102     // Handle "", $(null), or $( undefined )
103     if ( !selector ) {
104         return this;
105     }
106

```

3. 参数 selector 是 DOM 元素

如果参数 selector 有属性 nodeType，则认为 selector 是 DOM 元素，手动设置第一个元素和属性 context 指向该 DOM 元素、属性 length 为 1，然后返回包含了该 DOM 元素引用的 jQuery 对象。相关代码如下所示：

```

107     // Handle $(DOMElement)
108     if ( selector.nodeType ) {
109         this.context = this[0] = selector;
110         this.length = 1;
111         return this;
112     }
113

```

第 108 行：属性 nodeType 声明了文档树中节点的类型，例如，Element 节点的该属性值是 1，Text 节点是 3，Comment 节点是 9，Document 对象是 9，DocumentFragment 节点是 11。

4. 参数 selector 是字符串 “body”

如果参数 selector 是字符串 “body”，手动设置属性 context 指向 document 对象、第一个元素指向 body 元素、属性 length 为 1，最后返回包含了 body 元素引用的 jQuery 对象。这里是对查找字符串 “body”的优化，因为文档树中只会存在一个 body 元素。相关代码如下所示：

```

114     // The body element only exists once, optimize finding it
115     if ( selector === "body" && !context && document.body ) {
116         this.context = document;
117         this[0] = document.body;
118         this.selector = selector;
119         this.length = 1;
120         return this;
121     }
122

```

5. 参数 selector 是其他字符串

如果参数 selector 是其他字符串，则先检测 selector 是 HTML 代码还是 #id。相关代码如下所示：

```

123     // Handle HTML strings
124     if ( typeof selector === "string" ) {
125         // Are we dealing with HTML string or an ID?
126         if ( selector.charAt(0) === "<" && selector.charAt( selector.length
- 1 ) === ">" && selector.length >= 3 ) {
127             // Assume that strings that start and end with <> are HTML and
skip the regex check
128             match = [ null, selector, null ];
129

```

```

130         } else {
131             match = quickExpr.exec( selector );
132         }
133     }

```

第126~128行：如果参数selector以“<”开头、以“>”结尾，且长度大于等于3，则假设这个字符串是HTML片段，跳过正则quickExpr的检查。注意这里仅仅是假设，并不一定表示它是真正合法的HTML代码，如“<div></p>”。

第131行：否则，用正则quickExpr检测参数selector是否是稍微复杂一些的HTML代码（如“abc<div>”）或#id，匹配结果存放在数组match中。正则quickExpr的定义如下：

```

39     // A simple way to check for HTML strings or ID strings
40     // Prioritize #id over <tag> to avoid XSS via location.hash (#9521)
41     quickExpr = /^(?:[^#<]*(<[\w\W]+>) [^>]*$|#[\w\-*]$)/,

```

正则quickExpr包含两个分组，依次匹配HTML代码和id。如果匹配成功，则数组match的第一个元素为参数selector，第二个元素为匹配的HTML代码或undefined，第三个元素为匹配的id或undefined。下面的例子测试了正则quickExpr的功能：

```

quickExpr.exec( '#target' );           // ["#target", undefined, "target"]
quickExpr.exec( '<div>' );           // ["<div>", "<div>", undefined]
quickExpr.exec( 'abc<div>' );        // ["abc<div>", "<div>", undefined]
quickExpr.exec( 'abc<div>abc#id' );    // ["abc<div>abc#id", "<div>", undefined]
quickExpr.exec( 'div' );               // null
quickExpr.exec( '<div><img></div>' ); // ["<div><img></div>", "<div><img></div>", undefined]

```

第41行黑底白字的#，在jQuery 1.6.3 和之后的版本中，为了避免基于location.hash的XSS攻击，于是在quickExpr中增加了#。在jQuery 1.6.3之前的版本中quickExpr的定义如下：

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

在jQuery 1.6.3 和之后的版本中，quickExpr匹配selector时如果遇到“#”，则认为不是HTML代码，而是#id，然后尝试调用document.getElementById()查找与之匹配的元素。而在jQuery 1.6.3之前的版本中，则只检查左尖括号和右尖括号，如果匹配则认为是HTML代码，并尝试创建DOM元素，这可能会导致恶意的XSS攻击。

假设有下面的场景：

在应用代码中出现\$(location.hash)，即根据location.hash的值来执行不同的逻辑，而用户可以自行在浏览器地址栏中修改hash值为“#”，并重新打开这个页面；此时\$(location.hash)在执行时变为\$(#)。在jQuery 1.6.3之前，“#”被认为是HTML代码并创建img元素，因为属性src指向的图片地址并不存在，事件句柄onerror被执行并弹出1。这样一来，攻击者就可以在事件句柄onerror中编写恶意的JavaScript代码，例如，读取用户cookie、发起Ajax请求等。

读者可以访问以下地址，查看更多相关信息：

<http://bugs.jquery.com/ticket/9521>

http://ma.la/jquery_xss/

(1) 参数 selector 是单独标签

如果参数 selector 是单独标签，则调用 document.createElement() 创建标签对应的 DOM 元素。相关代码如下所示：

```

134      //Verify a match, and that no context was specified for #id
135      if ( match && (match[1] || !context) ) {
136
137          // HANDLE: $(html) -> $(array)
138          if ( match[1] ) {
139              context = context instanceof jQuery ? context[0] : context;
140              doc = ( context ? context.ownerDocument || context : document );
141
142              // If a single string is passed in and it's a single tag
143              // just do a createElement and skip the rest
144              ret = rsingleTag.exec( selector );
145
146              if ( ret ) {
147                  if ( jQuery.isPlainObject( context ) ) {
148                      selector = [ document.createElement( ret[1] ) ];
149                      jQuery.fn.attr.call( selector, context, true );
150
151                  } else {
152                      selector = [ doc.createElement( ret[1] ) ];
153                  }
154

```

第 135 行：检测正则 quickExpr 匹配参数 selector 的结果，如果 match[1] 不是 undefined，即参数 selector 是 HTML 代码，或者 match[2] 不是 undefined，即参数 selector 是 #id，并且未传入参数 context。这行代码利用布尔表达式的计算顺序，省略了对 match[2] 的判断，完整的表达式如下：

```
if ( match && (match[1] || match[2] && !context) ) {
```

如果 match 不是 null 且 match[1] 是 undefined，那么此时 match[2] 必然不是 undefined，所以对 match[2] 的判断可以省略。

第 138 ~ 140 行：开始处理参数 selector 是 HTML 代码的情况，先修正 context、doc，然后用正则 rsingleTag 检测 HTML 代码是否是单独标签，匹配结果存放在数组 ret 中。正则 rsingleTag 的定义如下：

```

50      //Match a standalone tag
51      rsingleTag = /^<(\w+)\s*/\?>(?:<\/\1>)?$/,

```

正则 rsingleTag 包含一个分组 “(\w+)”，该分组中不包含左右尖括号、不能包含属性、可以自关闭或不关闭；“\1” 指向匹配的第一个分组 “(\w+)”。

第 146 ~ 153 行：如果数组 ret 不是 null，则认为参数 selector 是单独标签，调用 document.createElement() 创建标签对应的 DOM 元素；如果参数 context 是普通对象，则调用

jQuery方法.attr()并传入参数context，同时把参数context中的属性、事件设置到新创建的DOM元素上。

之所以把创建的DOM元素放入数组中，是为了在后面第160行方便地调用jQuery.merge()方法。方法jQuery.merge()用于合并两个数组的元素到第一个数组，相关内容在2.8.8节介绍和分析。

参数context的细节请参考2.1.1节；方法.attr()遇到特殊属性和事件类型属性时会执行同名的jQuery方法，相关内容将在8.2节介绍和分析；方法jQuery.isPlainObject()用于检测对象是否是“纯粹”的对象，即用对象直接量{}或new Object()创建的对象，这会在2.8.2节介绍和分析。

(2) 参数selector是复杂HTML代码

如果参数selector是复杂HTML代码，则利用浏览器的innerHTML机制创建DOM元素。相关代码如下所示：

```

155             } else {
156                 ret = jQuery.buildFragment( [ match[1] ], [ doc ] );
157                 selector = ( ret.cacheable ? jQuery.clone(ret.fragment):ret.
fragment ).childNodes;
158             }
159
160             return jQuery.merge( this, selector );
161

```

第156行：创建过程由方法jQuery.buildFragment()和jQuery.clean()实现，方法jQuery.buildFragment()返回值的格式为：

```

{
    fragment: 含有转换后的DOM元素的文档片段
    cacheable: HTML代码是否满足缓存条件
}

```

第157行：如果HTML代码满足缓存条件，则在使用转换后的DOM元素时，必须先复制一份再使用，否则可以直接使用。

方法jQuery.buildFragment()和jQuery.clean()将分别在2.4节和第2.5节中介绍和分析。

第160行：将新创建的DOM元素数组合并到当前jQuery对象中并返回。

(3) 参数selector是“#id”，且未指定参数context

如果参数selector是“#id”，且未指定参数context，则调用document.getElementById()查找含有指定id属性的DOM元素。相关代码如下所示：

```

162             // HANDLE: $("#id")
163             } else {
164                 elem = document.getElementById( match[2] );
165
166                 //Check parentNode to catch when Blackberry 4.6 returns
167                 //nodes that are no longer in the document #6963
168                 if ( elem && elem.parentNode ) {
169                     // Handle the case where IE and Opera return items

```

```

170          //by name instead of ID
171          if ( elem.id !== match[2] ) {
172              return rootjQuery.find( selector );
173          }
174
175          //Otherwise, we inject the element directly into the
176          //jQuery object
177          this.length = 1;
178          this[0] = elem;
179      }
180
181      this.context = document;
182      this.selector = selector;
183      return this;
184  }

```

第 162 ~ 164 行：如果参数 selector 是“#id”且未指定参数 context，则调用 document.getElementById() 查找含有指定 id 属性的 DOM 元素。

第 166 ~ 168 行：检查 parentNode 属性，因为 BlackBerry 4.6 会返回已经不在文档中的 DOM 节点。

第 169 ~ 173 行：如果所找到元素的属性 id 值与传入的值不相等，则调用 Sizzle 查找并返回一个含有选中元素的新 jQuery 对象。即使是 document.getElementById() 这样核心的方法也需要考虑浏览器兼容问题，在 IE 6、IE 7、某些版本的 Opera 中，可能会按属性 name 查找而不是 id。例如，下面的 HTML 代码，通过 document.getElementById() 并不能找到正确的 DOM 元素：

```

<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="description" content="head meta description">
</head>
<body>
    <div id="description">
        body div description
    </div>
    <form name="divId">
        <div id="divId"></div>
    </form>
    <script>
        alert( document.getElementById( 'description' ).outerHTML );
        alert( document.getElementById( 'divId' ).outerHTML );
    </script>
</body>
</html>

```

在 IE7 中的运行结果如图 2-2 和图 2-3 所示。

在这种情况下，Sizzle 先通过 document.getElementsByTagName("*") 取出所有的 DOM 元素，然后检查每个元素的属性 id 是否与指定值相等，如果相等，则放入返回结果中。具体可查阅第 3 章关于“Sizzle”的介绍和分析。

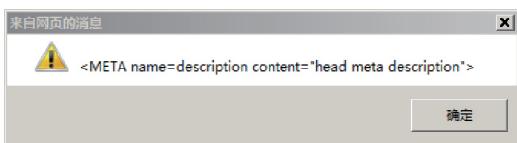


图 2-2 在 IE 7 中执行“alert(document.getElementById('description').outerHTML);”

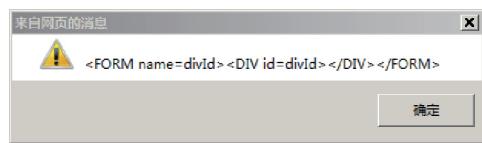


图 2-3 在 IE 7 中执行“alert(document.getElementById('divId').outerHTML);”

第 175 ~ 182 行：如果所找到元素的属性 id 值与传入的值相等，则设置第一个元素、属性 length、context、selector，并返回当前 jQuery 对象。

(4) 参数 selector 是选择器表达式

相关代码如下所示：

```

185      // HANDLE: $(expr, $(...))
186      } else if ( !context || context.jquery ) {
187          return ( context || rootjQuery ).find( selector );
188
189      // HANDLE: $(expr, context)
190      // (which is just equivalent to: $(context).find(expr)
191      } else {
192          return this.constructor( context ).find( selector );
193      }
194

```

此时依然在字符串分支中，参数 selector 不是单独标签、复杂 HTML 代码、#id，而是选择器表达式。如果没有指定上下文，则执行 rootjQuery.find(selector)；如果指定了上下文，且上下文是 jQuery 对象，则执行 context.find(selector)；如果指定了上下文，但上下文不是 jQuery 对象，则执行 this.constructor(context).find(selector)，即先创建一个包含了 context 的 jQuery 对象，然后在该 jQuery 对象上调用方法 .find()。

6. 参数 selector 是函数

相关代码如下所示：

```

195      // HANDLE: $(function)
196      // Shortcut for document ready
197      } else if ( jQueryisFunction( selector ) ) {
198          return rootjQuery.ready( selector );
199      }
200

```

第 197 ~ 199 行：如果参数 selector 是函数，则认为是绑定 ready 事件。从第 198 行代码可以看出 \$(function) 是 \$(document).ready(function) 的简写。

方法 jQueryisFunction() 将在 2.8.2 节介绍和分析。

7. 参数 selector 是 jQuery 对象

相关代码如下所示：

```

201      if ( selector.selector !== undefined ) {
202          this.selector = selector.selector;

```

```

203         this.context = selector.context;
204     }
205

```

第 201 ~ 204 行：如果参数 selector 含有属性 selector，则认为它是 jQuery 对象，将会复制它的属性 selector 和 context。而且在紧随其后的第 206 行会把参数 selector 中包含的选中元素引用，全部复制到当前 jQuery 对象中。

8. 参数 selector 是任意其他值

相关代码如下所示：

```

206     return jQuery.makeArray( selector, this );
207 },

```

第 206 行：如果 selector 是数组或伪数组（如 jQuery 对象），则都添加到当前 jQuery 对象中；如果 selector 是 JavaScript 对象，则作为第一个元素放入当前 jQuery 对象中；如果是其他类型的值，则作为第一个元素放入当前 jQuery 对象中。最后返回当前 jQuery 对象。

2.3.3 小结

至此，方法 jQuery.fn.init(selector, context, rootjQuery) 的 12 分支就介绍完了，相关的判断和执行过程可以整理为图 2-4。

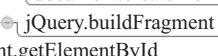
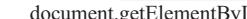
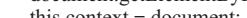
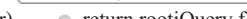
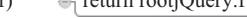
jQuery.fn.init(selector, context, rootjQuery)	
12 个分支	
	处理方式
!selector	return this;
selector.nodeType	this.context = this[0] = selector; this.length = 1; return this;
"body"	this.context = document; this[0] = document.body; this.selector = selector; this.length = 1; return this;
"string" 6 个分支	单独标签: <tag>  复杂 HTML 代码   #id  this.selector = selector; return this; \$(selector)  \$(selector, \$(...))  \$(selector, context) 
function	return rootjQuery.ready(selector);
\$(\$(...))	this.selector = selector.selector; this.context = selector.context;
任意其他值	return jQuery.makeArray(selector, this);

图 2-4 jQuery.fn.init(selector, context, rootjQuery) 的 12 个分支

2.4 jQuery.buildFragment(args, nodes, scripts)

2.4.1 实现原理

方法 `jQuery.buildFragment(args, nodes, scripts)` 先创建一个文档片段 `DocumentFragment`, 然后调用方法 `jQuery.clean(elems, context, fragment, scripts)` 将 HTML 代码转换为 DOM 元素, 并存储在创建的文档片段中。

文档片段 `DocumentFragment` 表示文档的一部分, 但不属于文档树。当把 `DocumentFragment` 插入文档树时, 插入的不是 `DocumentFragment` 自身, 而是它的所有子孙节点, 即可以一次向文档树中插入多个节点。当需要插入大量节点时, 相比于逐个插入节点, 使用 `documentFragment` 一次插入多个节点, 性能的提升会非常明显^①。

此外, 如果 HTML 代码符合缓存条件, 方法 `jQuery.buildFragment()` 还会把转换后的 DOM 元素缓存起来, 下次 (实际上是第三次) 转换相同的 HTML 代码时直接从缓存中读取, 不需要重复转换。

方法 `jQuery.buildFragment()` 同时为构造 jQuery 对象和 DOM 操作提供底层支持, DOM 操作将在第 11 章介绍和分析。

2.4.2 源码分析

方法 `jQuery.buildFragment(args, nodes, scripts)` 执行的 5 个关键步骤如下:

1) 如果 HTML 代码符合缓存条件, 则尝试从缓存对象 `jQuery.fragments` 中读取缓存的 DOM 元素。

2) 创建文档片段 `DocumentFragment`。

3) 调用方法 `jQuery.clean(elems, context, fragment, scripts)` 将 HTML 代码转换为 DOM 元素, 并存储在创建的文档片段中。

4) 如果 HTML 代码符合缓存条件, 则把转换后的 DOM 元素放入缓存对象 `jQuery.fragments`。

5) 最后返回文档片段和缓存状态 { `fragment: fragment, cacheable: cacheable` }。

下面来看看该方法的源码实现。

1. 定义 `jQuery.buildFragment(args, nodes, scripts)`

相关代码如下所示:

```
6085 jQuery.buildFragment = function( args, nodes, scripts ) {
```

第 6085 行: 定义方法 `jQuery.buildFragment(args, nodes, scripts)`, 它接受 3 个参数:

□ 参数 `args`: 数组, 含有待转换为 DOM 元素的 HTML 代码。

□ 参数 `nodes`: 数组, 含有文档对象、jQuery 对象或 DOM 元素, 用于修正创建文档片段 `DocumentFragment` 的文档对象。

□ 参数 `scripts`: 数组, 用于存放 HTML 代码中的 `script` 元素。方法 `jQuery.buildFragment()`

① <http://ejohn.org/blog/dom-documentfragments>。

会把该参数传给方法 `jQuery.clean()`，后者把 HTML 代码转换为 DOM 元素后，会提取其中的 `script` 元素并存入数组 `scripts`。在 11.2 节会看到，方法 `.domManip(args, table, callback)` 把转换后的 DOM 元素插入文档树后，会手动执行数组 `scripts` 中的元素。

2. 定义局部变量，修正文档对象 doc

相关代码如下所示：

```

6086     var fragment, cacheable, cacheresults, doc,
6087     first = args[ 0 ];
6088
6089     // nodes may contain either an explicit document object,
6090     // a jQuery collection or context object.
6091     // If nodes[0] contains a valid object to assign to doc
6092     if ( nodes && nodes[0] ) {
6093         doc = nodes[0].ownerDocument || nodes[0];
6094     }
6095
6096     // Ensure that an attr object doesn't incorrectly stand in as a document
object
6097     // Chrome and Firefox seem to allow this to occur and will throw exception
6098     // Fixes #8950
6099     if ( !doc.createDocumentFragment ) {
6100         doc = document;
6101     }
6102

```

第 6086 行：定义局部变量。变量 `fragment` 指向稍后可能创建的文档片段 `Document Fragment`；变量 `cacheable` 表示 HTML 代码是否符合缓存条件；变量 `cacheresults` 指向从缓存对象 `jQuery.fragments` 中取到的文档片段，其中包含了缓存的 DOM 元素；变量 `doc` 表示创建文档片段的文档对象。

第 6092 ~ 6101 行：修正文档对象 `doc`。数组 `nodes` 可能包含一个明确的文档对象，也可能包含 `jQuery` 对象或 DOM 元素，这里先尝试读取 `nodes[0]` 的属性 `ownerDocument` 并赋值给 `doc`，`ownerDocument` 表示 DOM 元素所在的文档对象。如果 `nodes[0].ownerDocument` 不存在，则假定 `nodes[0]` 为文档对象并赋值给 `doc`；但 `doc` 依然可能不是文档对象，如果调用 `jQuery` 构造函数时第二个参数是 `JavaScript` 对象，此时 `doc` 是传入的 `JavaScript` 对象而不是文档对象，例如，执行 “`$('abc<div></div>', {'class': 'test'})`” 时，`doc` 是 `{'class': 'test'}`，此时需要检查 `doc.createDocumentFragment` 是否存在，如果不存在则修正 `doc` 为当前文档对象 `document`。

3. 尝试从缓存对象 `jQuery.fragments` 中读取缓存的 DOM 元素

如果 HTML 代码符合缓存条件，则尝试从缓存对象 `jQuery.fragments` 中读取缓存的 DOM 元素，相关代码如下所示：

```

6103     // Only cache "small" (1/2 KB) HTML strings that are associated with the main
document
6104     // Cloning options loses the selected state, so don't cache them
6105     // IE 6 doesn't like it when you put <object> or <embed> elements in a

```

```

fragment
6106    // Also, WebKit does not clone 'checked' attributes on cloneNode, so don't
cache
6107    // Lastly, IE6,7,8 will not correctly reuse cached fragments that were
created from unknown elems #10501
6108    if ( args.length === 1 && typeof first === "string" &&
        first.length < 512 &&
        doc === document &&
6109        first.charAt(0) === "<" &&
        !rnocache.test( first ) &&
6110        (jQuery.support.checkClone || !rchecked.test( first )) &&
6111        (jQuery.support.html5Clone || !rnoshimcache.test( first )) ) {
6112        cacheable = true;
6113
6114        cacheresults = jQuery.fragments[ first ];
6115        if ( cacheresults && cacheresults !== 1 ) {
6116            fragment = cacheresults;
6117        }
6118    }
6119}
6120
6133 jQuery.fragments = {};

```

第6108 ~ 6111：HTML代码必须满足以下所有条件，才认为符合缓存条件：

- 数组args的长度为1，且第一个元素是字符串，即数组args中只含有一段HTML代码。
- HTML代码的长度小于512（1/2KB），否则可能会导致缓存占用的内存过大。
- 文档对象doc是当前文档对象，即只缓存为当前文档创建的DOM元素，不缓存其他框架（iframe）的。
- HTML代码以左尖括号开头，即只缓存DOM元素，不缓存文本节点。
- HTML代码中不能含有以下标签：`<script>`、`<object>`、`<embed>`、`<option>`、`<style>`。
- 当前浏览器可以正确地复制单选按钮和复选框的选中状态checked，或者HTML代码中的单选按钮和复选按钮没有被选中。
- 当前浏览器可以正确地复制HTML5元素，或者HTML代码中不含有HTML5标签。

HTML代码中不能含有的标签定义在正则rnocache中，该正则的定义代码如下所示：

```
5651 rnocache = /<(:script|object|embed|option|style)/i,
```

HTML代码中的单选按钮和复选框是否被选中通过正则rchecked检测，该正则的定义代码如下所示：

```
5653 // checked="checked" or checked
5654 rchecked = /checked\s*(?:[^=]|=\s*.checked.)/i,
```

HTML代码中是否含有HTML5标签通过正则rnoshimcache检测，该正则的定义代码如下所示：

```
5642 var nodeNames =
"abbr|article|aside|audio|canvas|datalist|details|figcaption|figure|footer|" +
```

```

5643           "header|hgroup|mark|meter|nav|output|progress|section|summary|
time|video",
5652   rnoshimcache = new RegExp( "<(?::" + nodeNames + ")" , "i" ),

```

测试项 `jQuery.support.checkClone` 指示当前浏览器是否可以正确地复制单选按钮和复选按钮的选中状态 `checked`, 请参见 7.2.14 节, 测试项 `jQuery.support.html5Clone` 指示当前浏览器是否可以正确地复制 HTML5 元素, 请参见 7.2.12 节。

第 6113 行: 如果 HTML 代码满足缓存条件, 则设置变量 `cacheable` 为 `true`。这是个很重要的状态值, 在使用转换后的 DOM 元素时, 如果变量 `cacheable` 为 `true`, 则必须先复制一份再使用, 否则可以直接使用。

第 6115 ~ 6118 行: 尝试从缓存对象 `jQuery.fragments` 中读取缓存的 DOM 元素。如果缓存命中, 并且缓存值不是 1, 则表示读取到的是文档片段, 赋值给变量 `fragment`, 文档片段中包含了缓存的 DOM 元素。稍后会看到对缓存对象 `jQuery.fragments` 中缓存值的分析。

4. 转换 HTML 代码为 DOM 元素

先创建一个文档片段 `DocumentFragment`, 然后调用方法 `jQuery.clean(elems, context, fragment, scripts)` 将 HTML 代码转换为 DOM 元素, 并存储在创建的文档片段中。相关代码如下所示:

```

6121   if ( !fragment ) {
6122     fragment = doc.createDocumentFragment();
6123     jQuery.clean( args, doc, fragment, scripts );
6124   }
6125

```

第 6121 行: 如果 `!fragment` 为 `true`, 表示需要执行转换过程。`!fragment` 为 `true` 时可能有三种情况:

- HTML 代码不符合缓存条件。
- HTML 代码符合缓存条件, 但此时是第一次转换, 不存在对应的缓存。
- HTML 代码符合缓存条件, 但此时是第二次转换, 对应的缓存值是 1。

第 6122 行: 调用原生方法 `document.createDocumentFragment()` 创建文档片段。

第 6123 行: 调用方法 `jQuery.clean()` 将 HTML 代码转换为 DOM 元素, 并存储在创建的文档片段中。该方法将在 2.5 节介绍和分析。

5. 把转换后的 DOM 元素放入缓存对象 `jQuery.fragments`

如果 HTML 代码符合缓存条件, 则把转换后的 DOM 元素放入缓存对象 `jQuery.fragments` 中。相关代码如下所示:

```

6126   if ( cacheable ) {
6127     jQuery.fragments[ first ] = cacheresults ? fragment : 1;
6128   }
6129

```

第 6126 ~ 6128 行: 如果 HTML 代码符合缓存条件, 则在缓存对象 `jQuery.fragments` 中对应的缓存值如表 2-2 所示。

表 2-2 符合缓存条件的 HTML 代码在缓存对象 `jQuery.fragments` 中对应的缓存值

场 景	转 换 前	转 换 后
第一次转换 HTML 代码	不存在	1
第二次转换同样的 HTML 代码	1	文档片段
两次以上转换同样的 HTML 代码	文档片段	文档片段

看到这里，可以把方法 `jQuery.buildFragment()` 的用法总结为：

- 如果 HTML 代码不符合缓存条件，则总是会执行转换过程。
- 如果 HTML 代码符合缓存条件，第一次转换后设置缓存值为 1，第二次转换后设置为文档片段，从第三次开始则从缓存中读取。

6. 返回文档片段和缓存状态 { fragment: fragment, cacheable: cacheable }

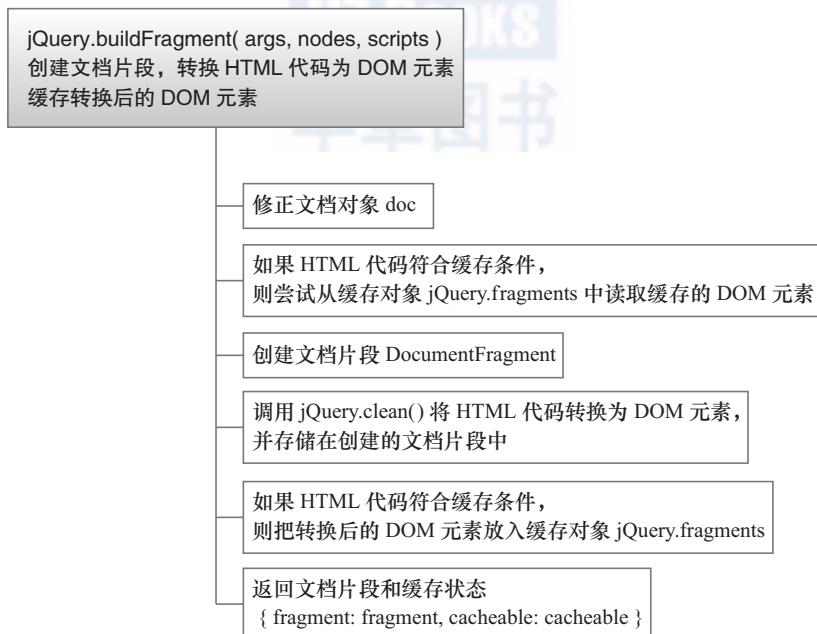
相关代码如下所示：

```
6130     return { fragment: fragment, cacheable: cacheable };
6131 };
```

第 6130 行：最后会返回一个包含了文档片段 `fragment` 和缓存状态 `cacheable` 的对象。文档片段 `fragment` 中包含了转换后的 DOM 元素，缓存状态 `cacheable` 则指示了如何使用这些 DOM 元素。

2.4.3 小结

方法 `jQuery.buildFragment(args, nodes, scripts)` 的执行过程可以总结为如图 2-5 所示。

图 2-5 `jQuery.buildFragment(args, nodes, scripts)` 的执行过程

2.5 jQuery.clean(elems, context, fragment, scripts)

2.5.1 实现原理

方法 `jQuery.clean(elems, context, fragment, scripts)` 负责把 HTML 代码转换成 DOM 元素，并提取其中的 `script` 元素。该方法先创建一个临时的 `div` 元素，并将其插入一个安全文档片段中，然后把 HTML 代码赋值给 `div` 元素的 `innerHTML` 属性，浏览器会自动生成 DOM 元素，最后解析 `div` 元素的子元素得到转换后的 DOM 元素。

安全文档片段指能正确渲染 HTML5 元素的文档片段，通过在文档片段上创建 HTML5 元素，可以教会浏览器正确地渲染 HTML5 元素，稍后的源码分析会介绍其实现过程。

如果 HTML 代码中含有需要包裹在父标签中的子标签，例如，子标签 `<option>` 需要包裹在父标签 `<select>` 中，方法 `jQuery.clean()` 会先在 HTML 代码的前后加上父标签和关闭标签，在设置临时 `div` 元素的 `innerHTML` 属性生成 DOM 元素后，再层层剥去包裹的父元素，取出 HTML 代码对应的 DOM 元素。

如果 HTML 代码中含有 `<script>` 标签，为了能执行 `<script>` 标签所包含的 JavaScript 代码或引用的 JavaScript 文件，在设置临时 `div` 元素的 `innerHTML` 属性生成 DOM 元素后，方法 `jQuery.clean()` 会提取其中的 `script` 元素放入数组 `scripts`。注意，将含有 `<script>` 标签的 HTML 代码设置给某个元素的 `innerHTML` 属性后，`<script>` 标签所包含的 JavaScript 代码不会自动执行，所引用的 JavaScript 文件也不会加载和执行。在 11.2.1 节分析 DOM 操作的核心工具方法 `jQuery.fn.domManip()` 时会看到，在生成的 DOM 元素插入文档树后，数组 `scripts` 中的 `script` 元素会被逐个手动执行。

2.5.2 源码分析

方法 `jQuery.clean(elems, context, fragment, scripts)` 执行的 8 个关键步骤如下：

- 1) 创建一个临时 `div` 元素，并插入一个安全文档片段中。
- 2) 为 HTML 代码包裹必要的父标签，然后赋值给临时 `div` 元素的 `innerHTML` 属性，从而将 HTML 代码转换为 DOM 元素，之后再层层剥去包裹的父元素，得到转换后的 DOM 元素。
- 3) 移除 IE 6/7 自动插入的空 `tbody` 元素，插入 IE 6/7/8 自动剔除的前导空白符。
- 4) 取到转换后的 DOM 元素集合。
- 5) 在 IE 6/7 中修正复选框和单选按钮的选中状态。
- 6) 合并转换后的 DOM 元素。
- 7) 如果传入了文档片段 `fragment`，则提取所有合法的 `script` 元素存入数组 `scripts`，并把其他元素插入文档片段 `fragment`。
- 8) 最后返回转换后的 DOM 元素数组。

下面来看看该方法的源码实现。

1. 定义 `jQuery.clean(elems, context, fragment, scripts)`

相关代码如下所示：

```
6256     clean: function( elems, context, fragment, scripts ) {
```

第6256行：定义方法jQuery.clean(elems, context, fragment, scripts)，它接受4个参数：

- 参数elems：数组，包含了待转换的HTML代码。
- 参数context：文档对象，该参数在方法jQuery.buildFragment()中被修正为正确的文档对象（变量doc），稍后会调用它的方法createTextNode()创建文本节点、调用方法createElement()创建临时div元素。
- 参数fragment：文档片段，作为存放转换后的DOM元素的占位符，该参数在jQuery.buildFragment()中被创建。
- 参数scripts：数组，用于存放转换后的DOM元素中的script元素。

2. 修正文档对象context

相关代码如下所示：

```
6257         var checkScriptType;
6258
6259         context = context || document;
6260
6261         // !context.createElement fails in IE with an error but returns typeof
'object'
6262         if ( typeof context.createElement === "undefined" ) {
6263             context = context.ownerDocument || context[0] && context[0].owner
Document || document;
6264         }
6265
```

第6258~6264行：修正文档对象context，与方法jQuery.buildFragment()对文档对象doc的修正类似，ownerDocument表示了DOM元素所在的文档对象。如果文档对象context没有createElement方法，则尝试读取context.ownerDocument或context[0].ownerDocument，如果没有，默认为当前文档对象document。

既然方法jQuery.buildFragment()已经谨慎地修正了文档对象doc，并传给了方法jQuery.clean()，那么这里为什么要再次做类似的修正呢？这是为了方便直接调用jQuery.clean()转换HTML代码为DOM元素。例如，在DOM操作模块的方法.before()和.after()中，将直接调用jQuery.clean()转换HTML代码为DOM元素，且只传入了待转换的HTML代码数组elems，而没有传入文档对象context、文档片段fragment和script元素数组scripts，相关代码如下所示：

```
// jQuery.fn.before()
5776     before: function() {
5782         var set = jQuery.clean( arguments );
5783
// jQuery.fn.after()
5788     after: function() {
5795         set.push.apply( set, jQuery.clean(arguments) );
```

方法jQuery.fn.before()在每个匹配元素之前插入指定的节点，方法jQuery.fn.after()则在每个匹配的元素之后插入指定的节点，这两个方法将在11.2.4节和11.2.5节介绍和分析。

3. 遍历待转换的 HTML 代码数组 elems

相关代码如下所示：

```

6266     var ret = [], j;
6267
6268     for ( var i = 0, elem; (elem = elems[i]) != null; i++ ) {
6269         if ( typeof elem === "number" ) {
6270             elem += "";
6271         }
6272
6273         if ( !elem ) {
6274             continue;
6275         }
6276
6277         // Convert html string into DOM nodes
6278         if ( typeof elem === "string" ) {

```

第 6266 行：数组 ret 用于存放转换后的 DOM 元素。

第 6268 行：开始遍历待转换的 HTML 代码数组。

注意，这里有个减少代码量的小技巧，在 for 语句的第 1 部分，声明了循环变量 elem，在 for 语句的第 2 部分取出 elems[i] 赋值给 elem，并判断 elem 的有效性。传统的做法可能是比较循环变量 i 与 elems.length，然后在 for 循环体中把 elems[i] 赋值给 elem，再判断 elem 的有效性。但这里通过一条 for 语句完成了循环变量 elem 的定义、赋值和有效性判断，减少了代码量，很实用且不影响阅读，读者可以在自己的代码中尝试应用这种写法。

另外，判断 elem 的有效性时使用的是“!=”，这样可以同时过滤 null 和 undefined，却又不会过滤整型数字 0。

第 6269 ~ 6271 行：如果 elem 是数值型，通过让 elem 自加一个空字符串，把 elem 转换为字符串，这也是一个很实用的小技巧。后面第 6280 行的 context.createTextNode() 也可以支持参数为数值型，这里把数值型转换为字符串，是为了简化随后对 elem 有效性和类型的判断。

第 6273 行：如果 !elem 为 true，即 elem 可以转换为 false，那么跳过本次循环，执行下一次循环。这行代码用于过滤空字符串的情况。如果 elem 是整型数字 0，因为在前面的代码中已经被转换成了字符串 “0”，所以这里可以简单地判断 !elem。

在上面的两个 if 语句中，即使 if 语句块中只有一行代码，仍然用花括号包裹起来，这是一个好的编码习惯，方便阅读和理解，避免潜在的错误。

第 6278 行：如果 elem 是字符串，即 HTML 代码，则开始转换 HTML 代码为 DOM 元素，这之后的 if 语句块是 jQuery.clean() 的核心代码。

(1) 创建文本节点

如果 HTML 代码中不包含标签、字符代码和数字代码，则调用原生方法 document.createTextNode() 创建文本节点，相关代码如下所示：

```

6279         if ( !rhtml.test( elem ) ) {

```

```

6280           elem = context.createTextNode( elem );
6281     } else {

```

第6279行：用正则rhtml检测HTML代码中是否含有标签、字符代码或数字代码，该正则的定义代码如下：

```
5649   rhtml = /<|&#?\w+;/,
```

标签的特征字符是左尖括号“<”，字符代码的特征字符是“&”，数字代码的特征字符是“&#”。字符代码和数字代码是特殊符号的两种形式，常见的特殊符号与字符代码、数字代码的编码对照表如表2-3所示。

表2-3 常见特殊符号与字符代码、数字代码的编码对照表

特 殊 符 号	字 符 代 码	数 字 代 码	备 注
”	"	"	双引号
'	'	'	单引号
<	<	<	小于
>	>	>	大于
&	&	&	
	 	 	空格
©	©	©	版权符号
®	®	®	注册符号
™	™	™	商标符号

完整的特殊符号编码对照表请访问：

<http://www.w3.org/MarkUp/Guide/Advanced.html>

http://www.w3schools.com/tags/ref_entities.asp

第6280行：原生方法document.createTextNode()用于创建文本节点，但是对于传给它的字符串参数不会做转义解析，也就是说，该方法不能正确地解析和创建包含了字符代码或数字代码的字符串，而浏览器的innerHTML机制则可以。例如，下面的代码将在页面中输出“©©”：

```

document.body.innerHTML = '&copy;';
// 显示转义后的 "@"

document.body.appendChild( document.createTextNode('&copy;') );
// 显示原始字符串 "&copy;"

```

第6281行：从这行代码开始转换包含了标签、字符代码或数字代码的HTML代码。

(2) 修正自关闭标签

相关代码如下所示：

```

6282           // Fix "XHTML"-style tags in all browsers
6283           elem = elem.replace(rxhtmlTag, "<$1></$2>");
6284

```

第6282~6283行：用正则rxhtmlTag匹配HTML代码中的自关闭标签，并通过方法

replace() 替换为成对的标签，例如，`<div/>`会被修正为`<div></div>`。自关闭标签是指没有对应的关闭标签，而是在标签的最后加一个 "/" 来关闭它，例如，`<div/>`。方法 replace() 的第二个参数中的 \$1 和 \$2 分别对应正则 rxhtmlTag 的第一个和第二个分组。

正则 rxhtmlTag 是修正自关闭标签的关键所在，它的定义代码如下：

```
5646     rxhtmlTag = /<(?![area|br|col|embed|hr|img|input|link|meta|param)
(([\w:]+)[^>]*)\/>/ig,
```

这个正则有些长，可以先用几个例子来测试一下它的功能。

例 1 最简单的自关闭标签。

```
'<div/>'.replace( rxhtmlTag, '<$1></$2>' );
// 输出: <div></div>
```

例 2 带有属性的自关闭标签。

```
'<div class="abc"/>'.replace( rxhtmlTag, '<$1></$2>' );
// 输出: <div class="abc"></div>
```

例 3 多个标签组合、标签前后有其他字符、大写标签`<A>`、不需要关闭的标签`<input>`、多行标签`\n`。

```
'a<div/>a \t b<A/>b \n c<xyz/>c \n d<IMG/>d \r\n e<input>e'.replace( rxhtmlTag,
'<$1></$2>' );
// 输出: "a<div></div>a \t b<A></A>b \n c<xyz></xyz>c \n d<IMG>d \r\n e<input>e"
```

现在来看看正则 rxhtmlTag 的精髓之处。

首先过滤掉不需要修正的标签：`(?!area|br|col|embed|hr|img|input|link|meta|param)`。`(?!p)`是反前向声明，要求接下来的字符不与模式 p 匹配。如果 HTML 代码中出现了这些标签，则不做任何处理。例如：

```
'<areadiv/>'.replace( rxhtmlTag, '<$1></$2>' )
// 输出: <areadiv/>
```

然后是精妙的嵌套分组，巧妙地提取了标签，同时又保留了属性：`<`和`/>`包围的`(([\w:]+)[^>]*)`作为第一个分组，其中包含了标签和属性；嵌套的`([\w:]+)`作为第二个分组，其中只包含了标签。

关于正则表达式和方法 replace() 的基础知识，不熟悉的读者请查阅相关的书籍资料。

(3) 创建一个临时 div 元素

相关代码如下所示：

```
6285                                     // Trim whitespace, otherwise indexOf won't work as
expected
6286                                     var tag = ( rtagName.exec( elem ) || [ "", "" ] )[1].
toLowerCase(),
6287                                     wrap = wrapMap[ tag ] || wrapMap._default,
6288                                     depth = wrap[0],
6289                                     div = context.createElement("div");
6290
```

第6286行：提取HTML代码中的标签部分，删除了前导空白符和左尖括号，并转换为小写赋值给变量wrap。正则rtagName的定义如下：

```
5647     rtagName = /<([\w:]+) /,
```

第6287行：从对象wrapMap中取出标签tag对应的父标签，它的定义和初始化代码如下：

```
5657     wrapMap = {
5658         option: [ 1, "<select multiple='multiple'>", "</select>" ],
5659         legend: [ 1, "<fieldset>", "</fieldset>" ],
5660         thead: [ 1, "<table>", "</table>" ],
5661         tr: [ 2, "<table><tbody>", "</tbody></table>" ],
5662         td: [ 3, "<table><tbody><tr>", "</tr></tbody></table>" ],
5663         col: [ 2, "<table><tbody></tbody><colgroup>", "</colgroup></table>" ],
5664         area: [ 1, "<map>", "</map>" ],
5665         _default: [ 0, "", "" ]
5666     },
5667     safeFragment = createSafeFragment( document );
5668
5669 wrapMap.optgroup = wrapMap.option;
5670 wrapMap.tbody = wrapMap.tfoot = wrapMap.colgroup = wrapMap.caption =
wrapMap.thead;
5671 wrapMap.th = wrapMap.td;
5672
5673 // IE can't serialize <link> and <script> tags normally
5674 if ( !jQuery.support.htmlSerialize ) {
5675     wrapMap._default = [ 1, "div<div>", "</div>" ];
5676 }
```

在对象wrapMap中包含了以下标签：option、optgroup、legend、thead、tbody、tfoot、colgroup、caption、tr、td、th、col、area、_default，每个标签对应一个数组，数组中的元素依次是：包裹的深度、包裹的父标签、父标签对应的关闭标签，例如，标签option对应的父标签是select，包裹深度为1。HTML语法规要求这些标签必须包含在其对应的父标签中，在随后设置临时div元素的innerHTML属性之前，会在便签前后自动加上父标签和关闭标签。

关于对象wrapMap，有几个有趣的地方需要注意一下：

- 标签option需要包含在多选的<select multiple='multiple'>中。因为如果包含在单选的<selector>中，创建的第一个option元素的selected属性会被浏览器默认设置为true；而如果包含在多选的<select multiple='multiple'>中，则不会被浏览器修改。可以用下面的代码验证：

```
// 创建一个临时div
var div = document.createElement('div');
// 单选<selector>, 输出 true
div.innerHTML = '<select><option>0</option></select>';
console.log( div.getElementsByName('option')[0].selected ); // true
// 多选<select multiple="multiple">, 输出 false
div.innerHTML = '<select multiple="multiple"><option>0</option></select>';
console.log( div.getElementsByName('option')[0].selected ); // false
```

- 严格按照HTML语法为tr、td、th添加父标签tbody。虽然设置innerHTML后浏览器

会为 tr、td、th 自动添加父元素 tbody，但保持结构良好的 HTML 代码是个好习惯。

□ 在 IE 9 以下的浏览器中，不能序列化标签<link> 和 <script>，即通过浏览器的 innerHTML 机制不能将其转换为对应的 link 元素和 script 元素，此时测试项 jQuery.support.htmlSerialize 为 false。解决方案是在标签<link> 和 <script> 外包裹一层元素再转换。包裹的元素定义在 wrapMap._default 中，_default 默认为 [0, "", ""]，如果 jQuery.support.htmlSerialize 为 false，则会在第 5675 行被修正为 [1, "div<div>", "</div>"]。可以运行用下面的代码来验证，运行结果如图 2-6 所示。

```
// IE 9 以下的浏览器
// 创建一个临时 div
var div = document.createElement('div');
// link 不包裹元素，输出 0
div.innerHTML = '<link rel="stylesheet" type="text/css" href="test.css" />';
console.log( div.getElementsByTagName("link").length ); // 0
// link 包裹元素，输出 1
div.innerHTML = 'div<div><link rel="stylesheet" type="text/css" href="test.css" /></div>';
console.log( div.getElementsByTagName("link").length ); // 1
// script 不包裹元素，输出 0
div.innerHTML = '<script></script>';
console.log( div.getElementsByTagName("script").length ); // 0
// script 包裹元素，输出 1
div.innerHTML = 'div<div><script></script></div>';
console.log( div.getElementsByTagName("script").length ); // 1
```

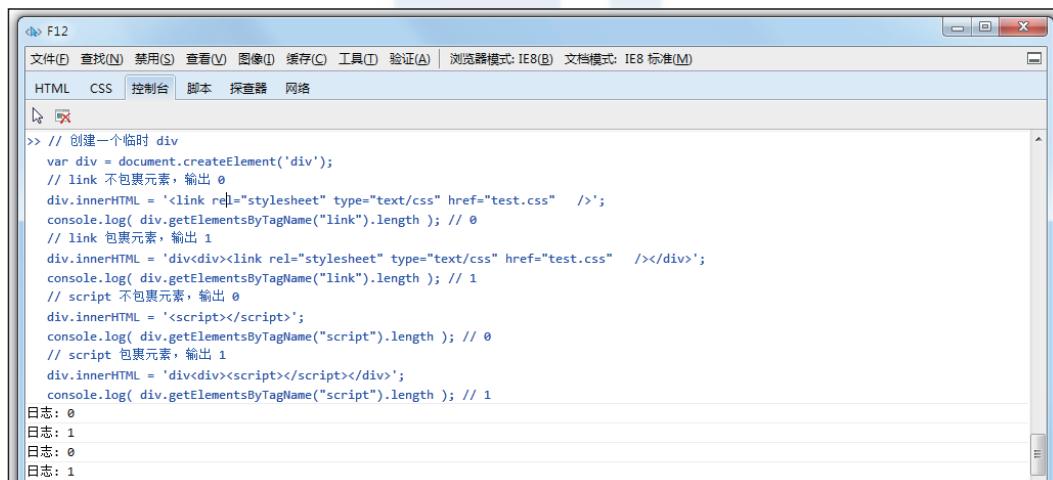


图 2-6 序列化标签<link> 和 <script>

对对象 wrapMap 和测试 jQuery.support.htmlSerialize 的分析到此为止，关于对象 wrapMap 中标签的含义和语法请参考相关的基础书籍，更多关于浏览器的测试和修正请参考第 7 章。

下面回到对方法 jQuery.clean() 源码的分析。

第 6288 行：取出被包裹的深度赋值给变量 depth，稍后将依据该变量层层剥去包裹的父元素。

第6289行：创建一个临时div元素，稍后它会被添加到一个安全文档片段中，并且它的innerHTML属性会被设置为待转换的HTML代码。

(4) 把临时div元素插入一个安全文档片段中

相关代码如下所示：

```

5291           // Append wrapper element to unknown element safe doc
fragment
5292           if ( context === document ) {
5293               // Use the fragment we've already created for this document
5294               safeFragment.appendChild( div );
5295           } else {
5296               // Use a fragment created with the owner document
5297               createSafeFragment( context ).appendChild( div );
5298           }
5299

```

第6292~6298行：如果传入的文档对象context是当前文档对象，则把临时div元素插入已创建的安全文档片段safeFragment中；否则，调用函数createSafeFragment()在文档对象context上创建一个新的安全文档片段，然后插入临时div元素。

所谓“安全”，是指不支持HTML5的浏览器也能够正确地解析和渲染未知的HTML5标签，即能够正确地构建DOM树，并且可以为之设置样式。IE 9以下的浏览器不支持HTML5，如果遇到未知标签（如<article>），浏览器会向DOM树中插入一个没有子元素的空元素。针对这个问题有一个“莫名其妙”的解决方法，就是在使用未知标签之前调用document.createElement('未知标签')创建一个对应的DOM元素，这样就可以“教会”浏览器正确地解析和渲染这个未知标签。

安全文档片段safeFragment在jQuery初始化时由函数createSafeFragment()创建，相关代码如下所示：

```

5667     safeFragment = createSafeFragment( document );
5668
5669 function createSafeFragment( document ) {
5670     var list = nodeNames.split( " | " ),
5671         safeFrag = document.createDocumentFragment();
5672
5673     if ( safeFrag.createElement ) {
5674         while ( list.length ) {
5675             safeFrag.createElement(
5676                 list.pop()
5677             );
5678         }
5679     }
5680     return safeFrag;
5681 }
5682
5683 var nodeNames = "abbr|article|aside|audio|canvas|datalist|details|figcaption|figure|footer|"
5684         + "header|hgroup|mark|meter|nav|output|progress|section|summary|time|video",
5685

```

变量 nodeNames 中存放了所有的 HTML5 标签，函数 createSafeFragment() 在传入的文档对象 document 上创建一个新的文档片段，然后在该文档片段上逐个创建 HTML5 元素，从而“教会”不支持 HTML5 的浏览器正确地解析和渲染 HTML5 标签。

(5) 利用浏览器的 innerHTML 机制将 HTML 代码转换为 DOM 元素

先为 HTML 代码包裹必要的父标签，然后赋值给临时 div 元素的 innerHTML 属性，从而将 HTML 代码转换为 DOM 元素，之后再层层剥去包裹的父元素，得到转换后的 DOM 元素。相关代码如下所示：

```

6300           // Go to html and back, then peel off extra wrappers
6301   div.innerHTML = wrap[1] + elem + wrap[2];
6302
6303           // Move to the right depth
6304   while ( depth-- ) {
6305       div = div.lastChild;
6306   }
6307

```

第 6300 行：为 HTML 代码包裹必要的父标签，然后赋值给临时 div 元素的 innerHTML 属性，浏览器会自动生成 DOM 元素。

第 6304 ~ 6306 行：用 while 循环层层剥去包裹的父元素，最终变量 div 将指向 HTML 代码对应的 DOM 元素的父元素。例如，标签 <option> 会被标签 <select> 包裹，包裹深度为 1，剥去一层后变量 div 指向了 select 元素。再例如，标签 <td> 会被 <table><tbody><tr> 包裹，包裹深度为 3，剥去 3 层后变量 div 指向了 tr 元素。如果 HTML 代码不需要包裹父标签，则变量 depth 为 0，不会进入 while 循环。

(6) 移除 IE 6/7 自动插入的空 tbody 元素

相关代码如下所示：

```

6308           // Remove IE's autoinserted <tbody> from table fragments
6309   if ( !jQuery.support.tbody ) {
6310
6311       // String was a <table>, *may* have spurious <tbody>
6312       var hasBody = tbody.test(elem),
6313           tbody = tag === "table" && !hasBody ?
6314               div.firstChild && div.firstChild.childNodes :
6315
6316               // String was a bare <thead> or <tfoot>
6317               wrap[1] === "<table>" && !hasBody ?
6318                   div.childNodes :
6319                   [];
6320
6321           for ( j = tbody.length - 1; j >= 0 ; --j ) {
6322               if ( jQuery.nodeName( tbody[ j ], "tbody" ) && !
tbody[ j ].childNodes.length ) {
6323                   tbody[ j ].parentNode.removeChild( tbody[ j ] );
6324               }
6325           }
6326       }
6327

```

第6309行：在IE 6/7中，浏览器会为空table元素自动插入空tbody元素，此时测试项jQuery.support.tbody为false。空元素指没有子元素的元素。

第6312行：用正则rtbody检查HTML代码中是否含有tbody标签，该正则的定义代码如下：

```
5648     rtbody = /<tbody/i,
```

第6313~6319行：提取IE 6/7自动插入的空tbody元素。这个复合三元表达式有些复杂，下面逐段分解之。

1) 如果执行“tag === "table" && !hasBody ? div.firstChild && div.firstChild.childNodes:”表示HTML代码中含有table标签，没有tbody标签，浏览器生成DOM元素时可能自动插入空tbody元素。此时变量div指向div元素，div.firstChild指向table元素，div.firstChild.childNodes则是tbody、thead、tfoot、colgroup、caption的元素集合。

2) 如果执行“wrap[1] === "<table>" && !hasBody ? div.childNodes:”表示为HTML代码包裹了父标签<table>，但是HTML代码中没有tbody标签，即HTML代码中含有thead、tfoot、colgroup、caption之一或多个，浏览器生成DOM元素时可能自动插入空tbody元素。此时变量div指向table元素，div.childNodes是tbody、thead、tfoot、colgroup、caption的元素集合。

3) []。如果HTML代码中含有tbody标签，无论空或非空都不需要删除，所以是[]。在其他情况下，浏览器生成DOM元素时不会自动插入空tbody元素，仍然是[]。

第6321~6326行：遍历数组tbody，移除空tbody元素，非空tbody元素不会移除。

第6322行：因为数组tbody中的元素还可能是thead、tfoot、colgroup、caption元素，所以要先判断tbody[j]是否是tbody元素；因为前面“提取IE 6/7自动插入的空tbody元素”的代码已经覆盖了所有可能的情况，所以!tbody[j].childNodes.length属于防御性检查，以防万一。

(7) 插入IE 6/7/8自动剔除的前导空白符

相关代码如下所示：

```
6328             // IE completely kills leading whitespace when innerHTML
is used
6329             if ( !jQuery.support.leadingWhitespace && rleading
Whitespace.test( elem ) ) {
6330                 div.insertBefore( context.createTextNode( rleading
Whitespace.exec(elem)[0] ), div.firstChild );
6331             }
6332
```

第6329行：在IE 6/7/8中，设置innerHTML属性时，浏览器会自动剔除前导空白符，测试测试项jQuery.support.leadingWhitespace为false。用正则rleadingWhitespace检测HTML代码中是否含有前导空白符，该正则的定义代码如下所示：

```
5645     rleadingWhitespace = /^\\s+/,
```

第6330行：用正则rleadingWhitespace提取HTML代码中的前导空白符，然后调用原生方法createTextNode()创建文本节点，最后插入div元素的第一个子元素前。

(8) 取到转换后的DOM元素集合

相关代码如下所示：

```

6333         elem = div.childNodes;
6334     }
6335 }
6336

```

(9) 在 IE 6/7 中修正复选框和单选按钮的选中状态

相关代码如下所示：

```

6337 // Resets defaultChecked for any radios and checkboxes
6338 // about to be appended to the DOM in IE 6/7 (#8060)
6339 var len;
6340 if ( !jQuery.support.appendChecked ) {
6341     if ( elem[0] && typeof (len = elem.length) === "number" ) {
6342         for ( j = 0; j < len; j++ ) {
6343             findInputs( elem[j] );
6344         }
6345     } else {
6346         findInputs( elem );
6347     }
6348 }
6349

```

第 6340 行：在 IE 6/7 中，复选框和单选按钮插入 DOM 树后，其选中状态 checked 会丢失，此时测试项 jQuery.support.appendChecked 为 false。通过在插入之前把属性 checked 的值赋值给属性 defaultChecked，可以解决这个问题。

第 6341 ~ 6344 行：遍历转换后的 DOM 元素集合，在每个元素上调用函数 findInputs(elem)。函数 findInputs(elem) 会找出其中的复选框和单选按钮，并调用函数 fixDefaultChecked(elem) 把属性 checked 的值赋值给属性 defaultChecked。

函数 findInputs() 和 fixDefaultChecked() 的相关代码如下所示：

```

6175 // Used in clean, fixes the defaultChecked property
6176 function fixDefaultChecked( elem ) {
6177     if ( elem.type === "checkbox" || elem.type === "radio" ) {
6178         elem.defaultChecked = elem.checked;
6179     }
6180 }
6181 // Finds all inputs and passes them to fixDefaultChecked
6182 function findInputs( elem ) {
6183     var nodeName = ( elem.nodeName || "" ).toLowerCase();
6184     if ( nodeName === "input" ) {
6185         fixDefaultChecked( elem );
6186     // Skip scripts, get other children
6187     } else if ( nodeName !== "script" && typeof elem.getElementsByName !== "undefined" ) {
6188         jQuery.grep( elem.getElementsByName("input"), fixDefaultChecked );
6189     }
6190 }
6191

```

(10) 合并转换后的 DOM 元素

相关代码如下所示：

```

6350 if ( elem.nodeType ) {

```

```

6351         ret.push( elem );
6352     } else {
6353         ret = jQuery.merge( ret, elem );
6354     }
6355 }
6356

```

第6355行：到此，数组elems中的所有HTML代码都已转换为DOM元素，并合并到了数组ret中。

4. 传入文档片段fragment的情况

如果传入了文档片段fragment，则遍历数组ret，提取所有（包括子元素）合法的script元素存入数组scripts，并把其他元素插入文档片段fragment。相关代码如下所示：

```

6357     if ( fragment ) {
6358         checkScriptType = function( elem ) {
6359             return !elem.type || rscriptType.test( elem.type );
6360         };
6361         for ( i = 0; ret[i]; i++ ) {
6362             if ( scripts && jQuery.nodeName( ret[i], "script" ) && (!ret[i].
6363 type || ret[i].type.toLowerCase() === "text/javascript") ) {
6364                 scripts.push( ret[i].parentNode ? ret[i].parentNode.removeChild(
6365                     ret[i] ) : ret[i] );
6366             } else {
6367                 if ( ret[i].nodeType === 1 ) {
6368                     var jsTags = jQuery.grep( ret[i].getElementsByTagName(
6369                         "script" ), checkScriptType );
6370                     ret.splice.apply( ret, [i + 1, 0].concat( jsTags ) );
6371                     fragment.appendChild( ret[i] );
6372                 }
6373             }
6374         }
6375

```

第6358~6360行：初始化变量checkScriptType为一个函数，用于检测script元素是否可执行。如果一个script元素没有指定属性type，或者属性type的值含有“/javascript”或“/ecmascript”，则认为是可执行的。正则rscriptType的定义代码如下：

```
5655     rscriptType = /\//(java|ecma)script/i,
```

第6361~6374行：遍历数组ret，提取所有（包括子元素）合法的script元素存入数组scripts，其他元素则插入文档片段fragment。

第6362~6363行：如果调用方法jQuery.clean()时传入了数组scripts，并找到了合法的script元素，则将该元素从其父元素中移除，然后存入数组scripts。如果一个script元素没有指定属性type，或者属性type的值是“text/javascript”，则认为是合法的。

通常应该为方法jQuery.clean()同时传入文档片段fragment和数组scripts。

第6366~6370行：在遍历数组ret的过程中，会提取当前元素所包含的script元素，并把其中可执行的插入数组ret，插入位置在当前元素之后，以便继续执行第6362~6363行的

检测和提取。script 元素是否可执行通过函数 checkScriptType(elem) 检测。

第 6371 行：在遍历数组 ret 的过程中，会把除了合法 script 元素之外的所有元素插入文档片段。

5. 返回转换后的 DOM 元素数组

相关代码如下所示：

```
6376     return ret;
6377 }
```

第 6376 行：最后，返回数组 ret，其中包含了所有转换后的 DOM 元素。但是要注意，如果传入了文档片段 fragment 和数组 scripts，那么调用 jQuery.clean() 的代码应该从文档片段 fragment 中读取转换后的 DOM 元素，并从数组 scripts 中读取合法的 script 元素；如果未传入，则只能使用返回值 ret。

2.5.3 小结

方法 jQuery.clean(elems, context, fragment, scripts) 的执行过程可以总结为图 2-7。

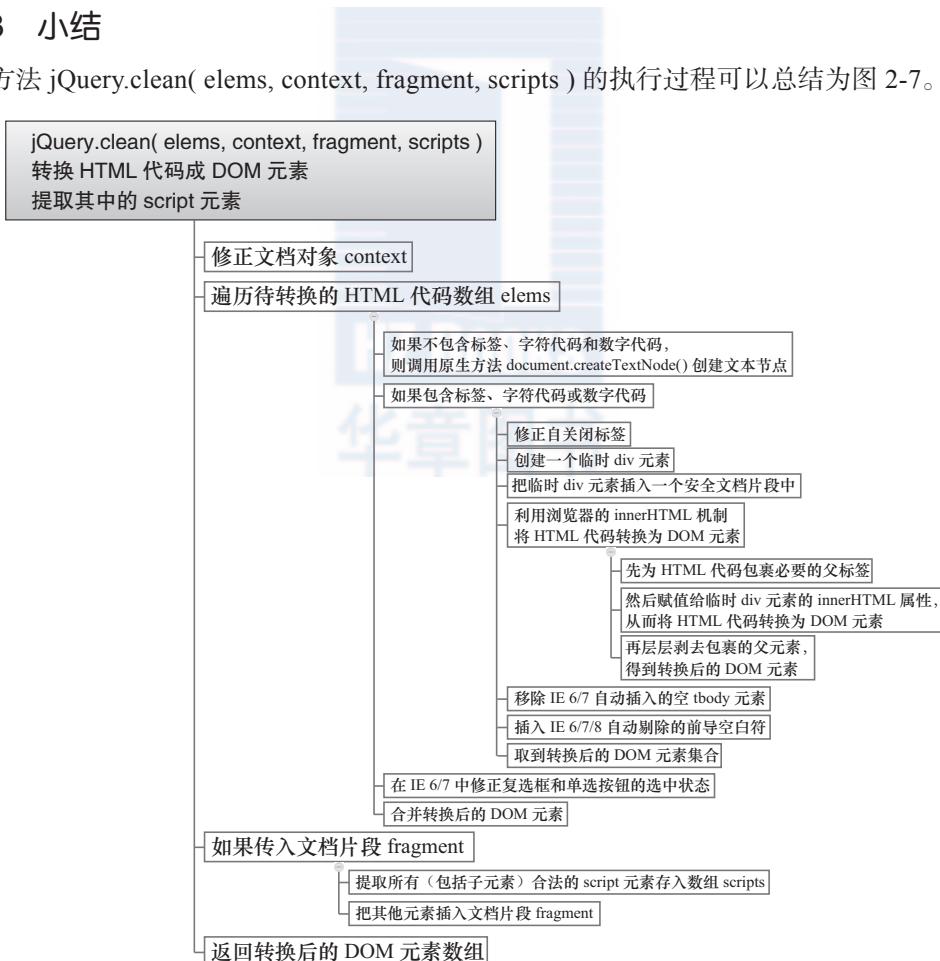


图 2-7 jQuery.clean(elems, context, fragment, scripts) 的执行过程

2.2 ~ 2.5节详细分析了构建jQuery对象的源码实现，从下一节开始，将介绍和分析其他的原型属性和方法，以及静态属性和方法，这些属性和方法是jQuery库的基础。

2.6 jQuery.extend()、jQuery.fn.extend()

2.6.1 如何使用

方法jQuery.extend()和jQuery.fn.extend()用于合并两个或多个对象的属性到第一个对象，它们的语法如下：

- `jQuery.extend([deep], target, object1 [, objectN])`
- `jQuery.fn.extend([deep], target, object1 [, objectN])`

其中，参数`deep`是可选的布尔值，表示是否进行深度合并（即递归合并）。合并行为默认是不递归的，如果第一个参数的属性本身是一个对象或数组，它会被第二个或后面的其他参数的同名属性完全覆盖。如果为`true`，表示进行深度合并，合并过程是递归的。

参数`target`是目标对象；参数`object1`和`objectN`是源对象，包含了待合并的属性。如果提供了两个或更多的对象，所有源对象的属性将会合并到目标对象；如果仅提供一个对象，意味着参数`target`被忽略，jQuery或jQuery.fn被当作目标对象，通过这种方式可以在jQuery或jQuery.fn上添加新的属性和方法，jQuery的其他模块大都是这么实现的。

方法jQuery.extend()和jQuery.fn.extend()常用于编写插件和处理函数的参数。

2.6.2 源码分析

方法jQuery.extend()和jQuery.fn.extend()执行的关键步骤如下所示：

- 1) 修正参数`deep`、`target`、源对象的起始下标。
- 2) 逐个遍历源对象：
 - a. 遍历源对象的属性。
 - b. 覆盖目标对象的同名属性；如果是深度合并，则先递归调用jQuery.extend()。

下面分析方法jQuery.extend()和jQuery.fn.extend()的源码。

1. 定义jQuery.extend()和jQuery.fn.extend()

相关代码如下所示：

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

第324行：因为参数的个数是不确定的，可以有任意多个，所以没有列出可接受的参数。

2. 定义局部变量

相关代码如下所示：

```
325     var options, name, src, copy, copyIsArray, clone,
326         target = arguments[0] || {},
327         i = 1,
328         length = arguments.length,
329         deep = false;
330
```

第 325 ~ 329 行：定义一组局部变量，它们的含义和用途如下：

- 变量 options：指向某个源对象。
- 变量 name：表示某个源对象的某个属性名。
- 变量 src：表示目标对象的某个属性的原始值。
- 变量 copy：表示某个源对象的某个属性的值。
- 变量 copyIsArray：指示变量 copy 是否是数组。
- 变量 clone：表示深度复制时原始值的修正值。
- 变量 target：指向目标对象。
- 变量 i：表示源对象的起始下标。
- 变量 length：表示参数的个数，用于修正变量 target。
- 变量 deep：指示是否执行深度复制，默认为 false。

3. 修正目标对象 target、源对象起始下标 i

相关代码如下所示：

```

331     // Handle a deep copy situation
332     if ( typeof target === "boolean" ) {
333         deep = target;
334         target = arguments[1] || {};
335         // skip the boolean and the target
336         i = 2;
337     }
338
339     // Handle case when target is a string or something (possible in deep copy)
340     if ( typeof target !== "object" && !jQueryisFunction(target) ) {
341         target = {};
342     }
343
344     // extend jQuery itself if only one argument is passed
345     if ( length === i ) {
346         target = this;
347         --i;
348     }
349

```

第 331~337 行：如果第一个参数是布尔值，则修正第一个参数为 deep，修正第二个参数为目标对象 target，并且期望源对象从第三个元素开始。

变量 i 的初始值为 1，表示期望源对象从第 2 个元素开始；当第一个参数为布尔型时，变量 i 变为 2，表示期望源对象从第 3 个元素开始。

第 339 ~ 342 行：如果目标对象 target 不是对象、不是函数，而是一个字符串或其他的基本类型，则统一替换为空对象 {}，因为在基本类型上设置非原生属性是无效的。例如：

```

var s = 'hi';
s.test = 'hello';
console.log( s.test );    // 打印 undefined

```

第 344 ~ 348 行：变量 i 表示源对象开始的下标，变量 length 表示参数个数，如果二者

相等，表示期望的源对象没有传入，则把jQuery或jQuery.fn作为目标对象，并且把源对象的开始下标减一，从而使得传入的对象被当作源对象。变量length等于i可能有两种情况：

- extend(object)，只传入了一个参数。
- extend(deep, object)，传入了两个参数，第一个参数是布尔值。

4. 逐个遍历源对象

相关代码如下所示：

```
350     for ( ; i < length; i++ ) {
```

第350行：循环变量i表示源对象开始的下标，很巧妙的用法。

(1) 遍历源对象的属性

下面的代码用于遍历源对象的属性：

```
351         // Only deal with non-null/undefined values
352         if ( (options = arguments[ i ]) != null ) {
353             //Extend the base object
354             for ( name in options ) {
```

第352行：arguments是一个类似数组的对象，包含了传入的参数，可以通过整型下标访问指定位置的参数。这行代码把获取源对象和对源对象的判断合并为一条语句，只有源对象不是null、undefined时才会继续执行。

第353行：开始遍历单个源对象的属性。

(2) 覆盖目标对象的同名属性

相关代码如下所示：

```
355             src = target[ name ];
356             copy = options[ name ];
357
358             // Prevent never-ending loop
359             if ( target === copy ) {
360                 continue;
361             }
362
363             // Recurse if we're merging plain objects or arrays
364             if ( deep && copy && ( jQuery.isPlainObject(copy) || (copyisArray =
jQuery.isArray(copy)) ) ) {
365                 if ( copyisArray ) {
366                     copyisArray = false;
367                     clone = src && jQuery.isArray(src) ? src : [];
368
369                 } else {
370                     clone = src && jQuery.isPlainObject(src) ? src : {};
371                 }
372
373                 // Never move original objects, clone them
374                 target[ name ] = jQuery.extend( deep, clone, copy );
375
376                 // Don't bring in undefined values
377             } else if ( copy !== undefined ) {
378                 target[ name ] = copy;
```

```

379
380
381
382
383

```

第 355 ~ 361 行：变量 `src` 是原始值，变量 `copy` 是复制值。如果复制值 `copy` 与目标对象 `target` 相等，为了避免深度遍历时死循环，因此不会覆盖目标对象的同名属性。如果注释掉第 360 行，下面的代码会抛出堆栈溢出异常：

```

var o = {};
o.n1 = o;
$.extend( true, o, { n2: o } );
// 抛出异常：
// Uncaught RangeError: Maximum call stack size exceeded

```

注意，判断 `target === copy` 时使用的是 “`==`”，强制不做类型转换；如果使用 “`=`”，则可能因自动类型转换而导致错误^①。

第 364 ~ 374 行：如果是深度合并，且复制值 `copy` 是普通 JavaScript 对象或数组，则递归合并。

第 365 ~ 371 行：复制值 `copy` 是数组时，如果原始值 `src` 不是数组，则修正为空数组；复制值 `copy` 是普通 JavaScript 对象时，如果原始值 `src` 不是普通 JavaScript 对象，则修正为空对象 {}。把原始值 `src` 或修正后的值赋值给原始值副本 `clone`。

通过调用方法 `jQuery.isPlainObject(copy)` 判断复制值 `copy` 是否是“纯粹”的 JavaScript 对象，只有通过对象直接量 {} 或 `new Object()` 创建的对象，才会返回 `true`。例如：

```

jQuery.isPlainObject( { hello: 'world' } );      // true
jQuery.isPlainObject( new Object() );            // true
jQuery.isPlainObject( new Object(1) );           // true

```

方法 `jQuery.isPlainObject()` 将在 2.8.2 节介绍和分析。

第 374 行：先把复制值 `copy` 递归合并到原始值副本 `clone` 中，然后覆盖目标对象的同名属性。

第 376 ~ 379 行：如果不是深度合并，并且复制值 `copy` 不是 `undefined`，则直接覆盖目标对象的同名属性。

2.7 原型属性和方法

在构造 jQuery 对象模块时，除了 2.3 节和 2.6 节已经介绍和分析的 `jQuery.fn.init()` 和 `jQuery.fn.extend()` 外，还定义了一些其他的原型属性和方法，其整体源码结构如代码清单 2-2 所示。

代码清单 2-2 原型属性和方法

```

97  jQuery.fn = jQuery.prototype = {
98      constructor: jQuery,
99      init: function( selector, context, rootjQuery ) {}

```

^① <http://bugs.jquery.com/ticket/1907>。

```

210     selector: "",
213     jquery: "1.7.1",
216     length: 0,
219     size: function() {},
223     toArray: function() [],
229     get: function( num ) [],
241     pushStack: function( elems, name, selector ) {},
270     each: function( callback, args ) {},
274     ready: function( fn ) {}, //
284     eq: function( i ) {},
291     first: function() {},
295     last: function() {},
299     slice: function() {},
304     map: function( callback ) {},
310     end: function() {},
316     push: push,
317     sort: [].sort,
318     splice: [].splice
319 };

```

其中`.ready()`用于绑定`ready`事件，将在9.11节进行介绍和分析。下面对其他的原型属性和方法逐一介绍和分析。

2.7.1 .selector、.jquery、.length、.size()

属性`selector`用于记录jQuery查找和过滤DOM元素时的选择器表达式，但不一定是可执行的选择器表达式，该属性更多的是为了方便调试。下面是一些示例：

```

$('div').find('p').selector           // "div p"
$('div p').selector                  // "div p"
$('div').first().selector            // "div.slice(0,1)"

```

属性`jquery`表示正在使用的jQuery版本号。

属性`.length`表示当前jQuery对象中元素的个数。

方法`.size()`返回当前jQuery对象中元素的个数。方法`.size()`在功能上等价于属性`.length`，但应该优先使用属性`.length`，因为它没有函数调用开销。

属性`.selector`、`.jquery`、`.length`、`.size()`的相关代码如下所示：

```

209     // Start with an empty selector
210     selector: "",
211
212     // The current version of jQuery being used
213     jquery: "1.7.1",
214
215     // The default length of a jQuery object is 0
216     length: 0,
217
218     // The number of elements contained in the matched element set
219     size: function() {
220         return this.length;
221     },
222

```

2.7.2 .toArray()、.get([index])

1. .toArray()

方法 `.toArray()` 将当前 jQuery 对象转换为真正的数组，转换后的数组包含了所有元素。

方法 `.toArray()` 的实现巧妙地借用了数组的方法 `slice()`，相关的代码如下所示：

```

86     // Save a reference to some core methods
87     toString = Object.prototype.toString,
88     hasOwn = Object.prototype.hasOwnProperty,
89     push = Array.prototype.push,
90     slice = Array.prototype.slice,
91     trim = String.prototype.trim,
92     indexOf = Array.prototype.indexOf,

223     toArray: function() {
224         return slice.call( this, 0 );
225     },

```

第 86 ~ 92 行：连同 `slice()` 一起声明的还有 `toString()`、`hasOwn()`、`trim()`、`indexOf()`，这里通过声明对这些核心方法的引用，使得在 jQuery 代码中可以借用这些核心方法的功能，执行时可通过方法 `call()` 和 `apply()` 指定方法执行的环境，即关键字 `this` 所引用的对象。这种“借鸡下蛋”的技巧非常值得借鉴。

数组方法 `slice()` 返回数组的一部分，语法如下：

```

array.slice(start, end)
// 参数 start 表示数组片段开始处的下标。如果是负数，它声明从数组末尾开始算起的位置
// 参数 end 表示数组片段结束处的后一个元素的下标。如果没有指定这个参数，切分的数组包含从 start
开始到数组结束的所有元素。如果这个参数是负数，它声明的是从数组尾部开始算起的元素

```

2. .get([index])

方法 `.get([index])` 返回当前 jQuery 对象中指定位置的元素或包含了全部元素的数组。如果没有传入参数，则调用 `.toArray()` 返回包含了所有元素的数组；如果指定了参数 `index`，则返回一个单独的元素；参数 `index` 从 0 开始计算，并且支持负数，负数表示从元素集合末尾开始计算。相关代码如下所示：

```

227     // Get the Nth element in the matched element set OR
228     // Get the whole matched element set as a clean array
229     get: function( num ) {
230         return num == null ?
231             // Return a 'clean' array
232             this.toArray() :
233
234             // Return just the object
235             ( num < 0 ? this[ this.length + num ] : this[ num ] );
236     },

```

第 236 行：先判断 `num` 是否小于 0，如果小于 0，则用 `length + num` 重新计算下标，然后使用数组访问操作符 `[]` 获取指定位置的元素，这是支持下标为负数的一个小技巧；如果

num 大于等于 0，则直接返回指定位置的元素。

2.7.3 .each(function(index, Element))、jQuery.each(collection, callback(indexInArray, valueOfElement))

1. .each(function(index, Element))

方法 .each() 遍历当前 jQuery 对象，并在每个元素上执行回调函数。每当回调函数执行时，会传递当前循环次数作为参数，循环次数从 0 开始计数；更重要的是，回调函数是在当前元素为上下文的语境中触发的，即关键字 this 总是指向当前元素；在回调函数中返回 false 可以终止遍历。

方法 .each() 内部通过简单的调用静态方法 jQuery.each() 实现，相关代码如下所示：

```
267 // Execute a callback for every element in the matched set.
268 // (You can seed the arguments with an array of args, but this is
269 // only used internally.)
270 each: function( callback, args ) {
271     return jQuery.each( this, callback, args );
272 },
```

2. jQuery.each(collection, callback(indexInArray, valueOfElement))

静态方法 jQuery.each() 是一个通用的遍历迭代方法，用于无缝地遍历对象和数组。对于数组和含有 length 属性的类数组对象（如函数参数对象 arguments），该方法通过下标遍历，从 0 到 length-1；对于其他对象则通过属性名遍历（for-in）。在遍历过程中，如果回调函数返回 false，则结束遍历。相关代码如下所示：

```
627 // args is for internal usage only
628 each: function( object, callback, args ) {
629     var name, i = 0,
630         length = object.length,
631         isObj = length === undefined || jQuery.isFunction( object );
632
633     if ( args ) {
634         if ( isObj ) {
635             for ( name in object ) {
636                 if ( callback.apply( object[ name ], args ) === false ) {
637                     break;
638                 }
639             }
640         } else {
641             for ( ; i < length; ) {
642                 if ( callback.apply( object[ i++ ], args ) === false ) {
643                     break;
644                 }
645             }
646         }
647
648     // A special, fast, case for the most common use of each
649     } else {
650         if ( isObj ) {
651             for ( name in object ) {
652                 if ( callback.call( object[ name ], name, object[ name ] ) === false ) {
653                     break;
654                 }
655             }
656         }
657     }
658 }
```

```

655         }
656     } else {
657         for ( ; i < length; ) {
658             if ( callback.call( object[ i ], i, object[ i++ ] ) === false ) {
659                 break;
660             }
661         }
662     }
663 }
664
665     return object;
666 },

```

第 628 行：定义方法 `jQuery.each(object, callback, args)`，它接受 3 个参数。

- 参数 `object`: 待遍历的对象或数组。
- 参数 `callback`: 回调函数，会在数组的每个元素或对象的每个属性上执行。
- 参数 `args`: 传给回调函数 `callback` 的参数数组，可选。如果没有传入参数 `args`，则执行回调函数时会传入两个参数（下标或属性名，对应的元素或属性值）如果传入了参数 `args`，则只把该参数传给回调函数。

第 631 行：变量 `isObj` 表示参数 `object` 是对象还是数组，以便决定遍历方式。如果 `object.length` 是 `undefined` 或 `object` 是函数，则认为 `object` 是对象，设置变量 `isObj` 为 `true`，将通过属性名遍历；否则认为是数组或类数组对象，设置变量 `isObj` 为 `false`，将通过下标遍历。

第 633 ~ 646 行：如果传入了参数 `args`，对于对象，通过 `for-in` 循环遍历属性名，对于数组或类数组对象，则通过 `for` 循环遍历下标，执行回调函数时只传入一个参数 `args`。

注意，执行回调函数 `callback` 时通过方法 `apply()` 指定 `this` 关键字所引用的对象，同时要求并假设参数 `args` 是数组，如果不是就会抛出 `TypeError`。

第 648 ~ 663 行：如果未传入参数 `args`，对于对象，通过 `for-in` 循环遍历属性名，对于数组或类数组对象，则通过 `for` 循环遍历下标，执行回调函数时传入两个参数：下标或属性名，对应的元素或属性值。

注意，执行回调函数 `callback` 时通过方法 `call()` 指定 `this` 关键字所引用的对象。

似乎 `jQuery.each()` 的代码略显啰嗦，因为第 633 ~ 646 行和第 648 ~ 663 行的代码相似度很高，只是触发回调函数的方式和参数不同，完全可以考虑把它们合并，在合并后的代码中根据变量 `isObj` 的值决定触发方式和参数，这样就可以减少一半的代码，但是，这也会导致在遍历过程中需要反复判断变量 `isObj` 的值。两权相较，方法 `jQuery.each()` 选择了“略显啰嗦的”代码来避免性能下降。

第 665 行：最后，返回传入的参数 `object`。方法 `.each()` 调用 `jQuery.each()` 时，把当前 `jQuery` 对象作为参数 `object` 传入，这里返回该参数，以支持链式语法。

2.7.4 `.map(callback(index, domElement))`、`jQuery.map(arrayOrObject, callback(value, indexOrKey))`

1. `.map(callback(index, domElement))`

方法 `.map()` 遍历当前 `jQuery` 对象，在每个元素上执行回调函数，并将回调函数的返回

值放入一个新jQuery对象中。该方法常用于获取或设置DOM元素集合的值。

执行回调函数时，关键字this指向当前元素。回调函数可以返回一个独立的数据项或数据项数组，返回值将被插入结果集中；如果返回一个数组，数组中的元素会被插入结果集；如果回调函数返回null或undefined，则不会插入任何元素。

方法.map()内部通过静态方法jQuery.map()和原型方法.pushStack()实现，相关代码如下所示：

```
304     map: function( callback ) {
305         return this.pushStack( jQuery.map(this, function( elem, i ) {
306             return callback.call( elem, i, elem );
307         }));
308     },
```

原型方法.pushStack()将在2.7.5节介绍和分析。

2. jQuery.map(arrayOrObject, callback(value, indexOrKey))

静态方法jQuery.map()对数组中的每个元素或对象的每个属性调用一个回调函数，并将回调函数的返回值放入一个新的数组中。执行回调函数时传入两个参数：数组元素或属性值，元素下标或属性名。关键字this指向全局对象window。回调函数的返回值会被放入新的数组中；如果返回一个数组，数组中将被扁平化后插入结果集；如果返回null或undefined，则不会放入任何元素。相关代码如下所示：

```
760     //arg is for internal usage only
761     map: function( elems, callback, arg ) {
762         var value, key, ret = [],
763             i = 0,
764             length = elems.length,
765             //jquery objects are treated as arrays
766             isArray = elems instanceof jQuery || length !== undefined && typeof
length === "number" && ( ( length > 0 && elems[ 0 ] && elems[ length -1 ] ) || length ===
0 || jQuery.isArray( elems ) );
767
768         //Go through the array, translating each of the items to their
769         if ( isArray ) {
770             for ( ; i < length; i++ ) {
771                 value = callback( elems[ i ], i, arg );
772
773                 if ( value != null ) {
774                     ret[ ret.length ] = value;
775                 }
776             }
777
778         // Go through every key on the object,
779         } else {
780             for ( key in elems ) {
781                 value = callback( elems[ key ], key, arg );
782
783                 if ( value != null ) {
784                     ret[ ret.length ] = value;
785                 }
786             }
787     }
```

```

787      }
788
789      // Flatten any nested arrays
790      return ret.concat.apply( [], ret );
791  },

```

第 761 行：定义方法 `jQuery.map(elems, callback, arg)`，它接受 3 个参数：

- 参数 `elems`：待遍历的数组或对象。
- 参数 `callback`：回调函数，会在数组的每个元素或对象的每个属性上执行。执行时传入两个参数：数组元素或属性值，元素下标或属性名。
- 参数 `arg`：仅限于 `jQuery` 内部使用。如果调用 `jQuery.map()` 时传入了参数 `arg`，则该参数会被传给回调函数 `callback`。

第 766 行：变量 `isArray` 表示参数 `elems` 是否是数组，以便决定遍历方式。如果为 `true`，将通过下标遍历；否则将通过属性名遍历。这行复合布尔表达式有些长，为了方便分析，将这行代码等价地格式化为下面的形式：

```

isArray = elems instanceof jQuery
  || length !== undefined && typeof length === "number"
    && ( ( length > 0 && elems[ 0 ] && elems[ length -1 ] )
      || length === 0
      || jQuery.isArray( elems ) );

```

如果 `elems` 是 `jQuery` 对象，则变量 `isArray` 为 `true`；如果 `elem.length` 是数值型，且满足以下条件之一，则变量 `isArray` 为 `true`：

- `length` 大于 0，且 `elems[0]` 存在，且 `elems[length -1]` 存在，即 `elems` 是一个类数组对象。
- `length` 等于 0。
- `elems` 是真正的数组。

第 769 ~ 776 行：对于数组或类数组对象，则通过 `for` 循环遍历下标，为每个元素执行回调函数 `callback`，执行时依次传入三个参数：元素、下标、`arg`。如果回调函数的返回值不是 `null` 和 `undefined`，则把返回值放入结果集 `ret`。

第 778 ~ 787 行：对于对象，则通过 `for-in` 循环遍历属性名，为每个属性值执行回调函数 `callback`，执行时依次传入三个参数：属性值、属性名、`arg`。如果回调函数的返回值不是 `null` 和 `undefined`，则把返回值放入结果集 `ret`。

第 790 行：最后在空数组 `[]` 上调用方法 `concat()` 扁平化结果集 `ret` 中的元素，并返回。

2.7.5 .pushStack(elements, name, arguments)

原型方法 `.pushStack()` 创建一个新的空 `jQuery` 对象，然后把 DOM 元素集合放入这个 `jQuery` 对象中，并保留对当前 `jQuery` 对象的引用。

原型方法 `.pushStack()` 是核心方法之一，它为以下方法提供支持：

- `jQuery` 对象遍历：`.eq()`、`.first()`、`.last()`、`.slice()`、`.map()`。
- DOM 查找、过滤：`.find()`、`.not()`、`.filter()`、`.closest()`、`.add()`、`.andSelf()`。

- DOM 遍历: .parent()、.parents()、.parentsUntil()、.next()、.prev()、.nextAll()、.prevAll()、.nextUnit()、.prevUnit()、.siblings()、.children()、.contents()。
- DOM 插入: jQuery.before()、jQuery.after()、jQuery.replaceWith()、.append()、.prepend()、.before()、.after()、.replaceWith()。

相关代码如下所示:

```

239     // Take an array of elements and push it onto the stack
240     // (returning the new matched element set)
241     pushStack: function( elems, name, selector ) {
242         // Build a new jQuery matched element set
243         var ret = this.constructor();
244
245         if ( jQuery.isArray( elems ) ) {
246             push.apply( ret, elems );
247
248         } else {
249             jQuery.merge( ret, elems );
250         }
251
252         // Add the old object onto the stack (as a reference)
253         ret.prevObject = this;
254
255         ret.context = this.context;
256
257         if ( name === "find" ) {
258             ret.selector = this.selector + ( this.selector ? " " : "" ) + selector;
259         } else if ( name ) {
260             ret.selector = this.selector + "." + name + "(" + selector + ")";
261         }
262
263         // Return the newly-formed element set
264         return ret;
265     },

```

第241行: 定义方法.push(elems, name, selector), 它接受3个参数:

- 参数 elems: 将放入新jQuery对象的元素数组(或类数组对象)。
 - 参数 name: 产生元素数组 elems 的jQuery方法名。
 - 参数 selector: 传给jQuery方法的参数, 用于修正原型属性.selector。
- 第243行: 构造一个新的空jQuery对象ret, this.constructor指向构造函数jQuery()。
- 第245~250行: 把参数elems合并到新jQuery对象ret中。如果参数elems是数组, 则借用数组方法push()插入, 否则调用方法jQuery.merge(first, second)合并。

第253行: 在新Query对象ret上设置属性prevObject, 指向当前jQuery对象, 从而形成一个链式栈。因此, 方法.pushStack()的行为还可以理解为, 构建一个新的jQuery对象并入栈, 新对象位于栈顶, 这也是该方法如此命名的原因所在。

第255行: 在新jQuery对象ret上设置属性prevObject, 指向当前jQuery对象的上下文, 后续的jQuery方法可能会用到这个属性。

第257~261行: 在新jQuery对象ret上设置属性selector, 该属性不一定是合法的选择

器表达式，更多的是为了方便调试，例如，下面的代码打印了调用方法`.pushStack()`之后属性`selector`的值：

```
console.log( $('div').eq(0).selector );
//div.slice(0,1)

console.log( $('div').first().selector );
//div.slice(0,1)

console.log( $('div').last().selector );
//div.slice(-1)

console.log( $('div').slice(0,9).selector );
//div.slice(0,9)

console.log( $('div').find('p').selector );
//div p

console.log( $('div').not('.cls').selector );
//div.not(.cls)

console.log( $('div').find('p').not('.cls').selector );
//div p.not(.cls)
```

第 264 行：最后返回新 jQuery 对象 ret。

2.7.6 .end()

方法`.end()`结束当前链条中最近的筛选操作，并将匹配元素集合还原为之前的状态。相关代码如下所示：

```
310     end: function() {
311         return this.prevObject || this.constructor(null);
312     },
```

第 311 行：返回前一个 jQuery 对象。如果属性`prevObject`不存在，则构建一个空的 jQuery 对象返回。

方法`.pushStack()`用于入栈，方法`.end()`则用于出栈。这两个方法可以像下面的例子这样使用：

```
$('ul.first').find('.foo')
.css('background-color', 'red')
.end().find('.bar')
.css('background-color', 'green')
.end();
```

2.7.7 .eq(index)、.first()、.last()、.slice(start [, end])

方法`.eq(index)`将匹配元素集合缩减为集合中指定位置的元素；方法`.first()`将匹配元素集合缩减为集合中的第一个元素；方法`.last()`将匹配元素集合缩减为集合中的最后一个元素；

方法`.slice(start [, end])`将匹配元素集合缩减为指定范围的子集。

方法`.first()`和`.last()`通过调用`.eq(index)`实现，`.eq(index)`则通过`.slice(start [, end])`实现，`.slice(start [, end])`则通过调用`.pushStack(elements, name, arguments)`实现，方法调用链为`.first/last() → .eq(index) → .slice(start [, end]) → .pushStack(elements, name, arguments)`。相关代码如下所示：

```

284     eq: function( i ) {
285         i = +i;
286         return i === -1 ?
287             this.slice( i ) :
288             this.slice( i, i + 1 );
289     },
290
291     first: function() {
292         return this.eq( 0 );
293     },
294
295     last: function() {
296         return this.eq( -1 );
297     },
298
299     slice: function() {
300         return this.pushStack( slice.apply( this, arguments ),
301             "slice", slice.call(arguments).join(",") );
302     },

```

第285行：如果参数`i`是字符串，则通过在前面加上一个加号把该参数转换为数值。

第300~301行：先借用数组方法`slice()`从当前jQuery对象中获取指定范围的子集（数组），再调用方法`.pushStack()`把子集转换为jQuery对象，同时通过属性`prevObject`保留了对当前jQuery对象的引用。

2.7.8 `.push(value, ...)`、`.sort([orderfunc])`、`.splice(start, deleteCount, value, ...)`

方法`.push(value, ...)`向当前jQuery对象的末尾添加新元素，并返回新长度，例如：

```
var foo = $(document);
foo.push( document.body ); // 2
```

方法`.sort([orderfunc])`对当前jQuery对象中的元素进行排序，可以传入一个比较函数来指定排序方式，例如^②。

```
var foo = $([33, 4, 1111, 222]);
foo.sort(); // [1111, 222, 33, 4]
foo.sort(function(a, b){
    console.log('orderfun', a, b );
    return a - b;
}) // [4, 33, 222, 1111]
```

方法`.splice(start, deleteCount, value, ...)`向当前jQuery对象中插入、删除或替换元素。

^② 《JavaScript权威指南 第5版 中文版》，核心JavaScript参考手册，Array.sort()，P604。

如果从当前 jQuery 对象中删除了元素，则返回含有被删除元素的数组。例如：

```
var foo = $('<div id="d1" /><div id="d2" /><div id="d3" />');
// [<div id="d1"></div>, <div id="d2"></div>, <div id="d3"></div>]
foo.splice( 1, 2 );
// [<div id="d2"></div>, <div id="d3"></div>]
```

方法 .push()、.sort()、.splice() 仅在内部使用，都指向同名的数组方法，因此它们的参数、功能和返回值与数组方法完全一致。相关代码如下所示：

```
314      // For internal use only.
315      // Behaves like an Array's method, not like a jQuery method.
316      push: push,
317      sort: [].sort,
318      splice: [].splice
319  };
```

2.7.9 小结

构造 jQuery 对象模块的原型属性和方法可以总结为图 2-8。

jQuery.fn = jQuery.prototype 原型属性和方法	
.constructor	指向构造函数 jQuery()
.init(selector, context, rootjQuery)	构造函数，解析参数 selector 和 context 的类型，并执行相应的逻辑，最后返回 jQuery.fn.init() 的实例
.selector	记录 jQuery 查找和过滤 DOM 元素时的选择器表达式
.jquery	正在使用的 jQuery 版本号
.length	jQuery 对象中元素的个数
.size()	返回当前 jQuery 对象中元素的个数
.toArray()	将当前 jQuery 对象转换为真正的数组
.get([index])	返回当前 jQuery 对象中指定位置的元素或包含了全部元素的数组
.pushStack(elements, name, arguments)	创建一个新的空 jQuery 对象，然后把 DOM 元素集合放入这个 jQuery 对象中，并保留对当前 jQuery 对象的引用
.each(function(index, Element))	遍历当前 jQuery 对象中的元素，并在每个元素上执行回调函数
.ready(handler)	绑定 ready 事件
.eq(index)	将匹配元素集合缩减为位于指定位置的新元素

图 2-8 构造 jQuery 对象模块的原型属性和方法

.first()	将匹配元素集合缩减为集合中的第一个元素
.last()	将匹配元素集合缩减为集合中的最后一个元素
.slice()	将匹配元素集合缩减为指定范围的子集
.map(callback(index, domElement))	遍历当前jQuery对象中的元素，并在每个元素上执行回调函数，将回调函数的返回值放入一个新的jQuery对象中
.end()	结束当前链条中最近的筛选操作，并将匹配元素集合还原为之前的状态
.push()	Array.prototype.push
.sort()	[] .sort
.splice()	[] .splice

图 2-8 (续)

2.8 静态属性和方法

在构造jQuery对象模块中还定义了一些重要的静态属性和方法，它们是其他模块实现的基础。其整体源码结构如代码清单2-3所示。

代码清单2-3 静态属性和方法

```

388 jQuery.extend({
389   noConflict: function( deep ) {},
402   isReady: false,
406   readyWait: 1,
409   holdReady: function( hold ) {},
418   ready: function( wait ) {},
444   bindReady: function() {},
492  isFunction: function( obj ) {},
496   isArray: Array.isArray || function( obj ) {},
501   isWindow: function( obj ) {},
505   isNumeric: function( obj ) {},
509   type: function( obj ) {},
515   isPlainObject: function( obj ) {},
544   isEmptyObject: function( obj ) {},
551   error: function( msg ) {},
555   parseJSON: function( data ) {},
581   parseXML: function( data ) {},
601   noop: function() {},
606   globalEval: function( data ) {},
619   camelCase: function( string ) {},
623   nodeName: function( elem, name ) {},
628   each: function( object, callback, args ) {},
669   trim: trim ? function( text ) {} : function( text ) {},
684   makeArray: function( array, results ) {},
702   inArray: function( elem, array, i ) {},
724   merge: function( first, second ) {},
744   grep: function( elems, callback, inv ) {},

```

```

761     map: function( elems, callback, arg ) {},
794     guid: 1,
798     proxy: function( fn, context ) {},
825     access: function( elems, key, value, exec, fn, pass ) {},
852     now: function() {},
858     uaMatch: function( ua ) {},
870     sub: function() {},
891     browser: {}
892 });

```

`jQuery.isReady`、`jQuery.readyWait`、`jQuery.holdReady()`、`jQuery.ready()`、`jQuery.bindReady()`用于支持 ready 事件，将在 9.11 节对它们进行介绍和分析；`jQuery.each()`和`jQuery.map()`已经在 2.7.3 节和 2.7.4 节介绍和分析过了；`jQuery.sub()`在 jQuery 源码中没有用到，并且不推荐使用，这里不做分析；下面对其他的静态属性和方法逐一介绍和分析。

2.8.1 `jQuery.noConflict([removeAll])`

方法`jQuery.noConflict([removeAll])`用于释放 jQuery 对全局变量`$`的控制权，可选的参数`removeAll`指示是否释放对全局变量`jQuery`的控制权。`$`仅仅是`jQuery`的别名，所有的功能没有`$`也能使用。

很多 JavaScript 库使用美元符`$`作为函数名或变量名，在使用`jQuery`的同时，如果需要使用另一个 JavaScript 库，可以调用`$.noConflict()`返回`$`给其他库。如果有必要（例如，在一个页面中使用多个版本的`jQuery`库，但很少有这样的必要），也可以释放全局变量`jQuery`的控制权，只需要给这个方法传入参数`true`即可。

方法`jQuery.noConflict([removeAll])`相关代码如下所示：

```

30     // Map over jQuery in case of overwrite
31     _jQuery = window.jQuery,
32
33     // Map over the $ in case of overwrite
34     _$ = window.$,

```



```

388 jQuery.extend({
389     noConflict: function( deep ) {
390         if ( window.$ === jQuery ) {
391             window.$ = _$;
392         }
393
394         if ( deep && window.jQuery === jQuery ) {
395             window.jQuery = _jQuery;
396         }
397
398         return jQuery;
399     },

```



```

9245 // Expose jQuery to the global object
9246 window.jQuery = window.$ = jQuery;

```

第 30 ~ 34 行：`jQuery` 初始化时，把可能存在的`window.jQuery`和`window.$`备份到局部

变量`_jQuery`和`_$`。

第390~392行：如果`window.$ === jQuery`，则设置`window.$`为初始化时备份的`_$`。也就是说，只有在当前jQuery库持有全局变量`$`的情况下，才会释放`$`的控制权给前一个JavaScript库。

从jQuery 1.6开始增加了对`window.$ === jQuery`的检测。如果不检测，则每次调用`jQuery.noConflict()`时都会释放`$`给前一个JavaScript库，当页面中有两个以上定义了`$`的JavaScript库时，对`$`的管理将会变得混乱。

第394~396行：如果参数`deep`为`true`，并且`window.jQuery === jQuery`，则设置`window.jQuery`为初始化时备份的`_jQuery`。也就是说，如果参数`deep`为`true`，只有在当前jQuery库持有全局变量`jQuery`的情况下，才会释放`jQuery`的控制权给前一个JavaScript库。

从jQuery 1.6开始增加了对`window.$ === jQuery`的检测。如果不检测，则每次调用`jQuery.noConflict(true)`时都会释放`jQuery`给前一个JavaScript库，当页面中有两个以上定义了`jQuery`的JavaScript库时，对`jQuery`的管理将会变得混乱。

2.8.2 类型检测：`jQueryisFunction(obj)`、`jQuery.isArray(obj)`、`jQuery.isWindow(obj)`、`jQuery.isNumeric(value)`、`jQuery.type(obj)`、`jQuery.isPlainObject(object)`、`jQuery.isEmptyObject(object)`

1. `jQueryisFunction(obj)`、`jQuery.isArray(obj)`

方法`jQueryisFunction(obj)`用于判断传入的参数是否是函数；方法`jQuery.isArray(obj)`用于判断传入的参数是否是数组。这两个方法的实现依赖于方法`jQuery.type(obj)`，通过判断`jQuery.type(obj)`返回值是否是“function”和“array”来实现。相关代码如下所示：

```
489 // See test/unit/core.js for details concerning isFunction.
490 // Since version 1.3, DOM methods and functions like alert
491 // aren't supported. They return false on IE (#2968).
492 isFunction: function( obj ) {
493     return jQuery.type(obj) === "function";
494 },
495
496 isArray: Array.isArray || function( obj ) {
497     return jQuery.type(obj) === "array";
498 },
499
```

2. `jQuery.type(obj)`

方法`jQuery.type(obj)`用于判断参数的内建JavaScript类型。如果参数是`undefined`或`null`，返回“`undefined`”或“`null`”；如果参数是JavaScript内部对象，则返回对应的字符串名称；其他情况一律返回“`object`”。相关代码如下所示：

```
509 type: function( obj ) {
510     return obj == null ?
511         String( obj ) :
512         class2type[ toString.call(obj) ] || "object";
513 },
```

第 510 ~ 511 行：如果参数 obj 是 undefined 或 null，通过 String(obj) 转换为对应的原始字符串“undefined”或“null”。

第 512 行：先借用 Object 的原型方法 toString() 获取 obj 的字符串表示，返回值的形式为 [object class]，其中的 class 是内部对象类，例如，Object.prototype.toString.call(true) 会返回 [object Boolean]；然后从对象 class2type 中取出 [object class] 对应的小写字符串并返回；如果未取到则一律返回“object”。原型方法 toString() 和对象 class2type 的定义和初始化源码如下所示：

```
87     toString = Object.prototype.toString,
94     // [[Class]] -> type pairs
95     class2type = {};
894 // Ppulate the class2type map
895 jQuery.each("Boolean Number String Function Array Date RegExp Object".
split(" "), function(i, name) {
896     class2type[ "[object " + name + "]" ] = name.toLowerCase();
897 });

```

对象 class2type 初始化后的结构为：

```
{
  "[object Array]": "array"
  "[object Boolean]": "boolean"
  "[object Date]": "date"
  "[object Function]": "function"
  "[object Number]": "number"
  "[object Object]": "object"
  "[object RegExp]": "regexp"
  "[object String]": "string"
}
```

3. jQuery.isWindow(obj)

方法 jQuery.isWindow(obj) 用于判断传入的参数是否是 window 对象，通过检测是否存在特征属性 setInterval 来实现，相关代码如下所示：

```
500     // A crude way of determining if an object is a window
501     isWindow: function( obj ) {
502         return obj && typeof obj === "object" && "setInterval" in obj;
503     },

```

在本书写作时发布的 jQuery 1.7.2 中，该方法改为检测特征属性 window，该属性是对窗口自身的引用，相关代码如下所示：

```
// 1.7.2
isWindow: function( obj ) {
    return obj != null && obj == obj.window;
},
```

4. jQuery.isNumeric(value)

方法 jQuery.isNumeric(value) 用于判断传入的参数是否是数字，或者看起来是否像数

字，相关代码如下所示：

```
505     isNumeric: function( obj ) {
506         return !isNaN( parseFloat(obj) ) && isFinite( obj );
507     },
```

第506行：先执行 `parseFloat(obj)` 尝试把参数 `obj` 解析为数字，然后用 `isNaN()` 判断解析结果是否是合法数字，并用 `isFinite()` 判断参数 `obj` 是否是有限的。如果 `parseFloat(obj)` 的解析结果是合法数字，并且参数 `obj` 是有限数字，则返回 `true`；否则返回 `false`。

方法 `parseFloat(string)` 用于对字符串参数进行解析，并返回字符串中的第一个数字。在解析过程中，如果遇到了不是有效数字的字符，解析就会停止并返回解析结果；如果字符串没有以一个有效的数字开头，则返回 `Nan`；如果传入的参数是对象，则自动调用该对象的方法 `toString()`，得到该对象的字符串表示，然后再执行解析过程。

方法 `isNaN(x)` 用于判断参数是否为非数字值，常用于检测方法 `parseFloat()` 和 `parseInt()` 的解析结果。

方法 `isFinite(number)` 用于判断一个数字是否是有限的。

5. jQuery.isPlainObject(object)

方法 `jQuery.isPlainObject(object)` 用于判断传入的参数是否是“纯粹”的对象，即是否是用对象直接量 {} 或 `new Object()` 创建的对象。相关代码如下所示：

```
515     isPlainObject: function( obj ) {
516         //Must be an Object.
517         //Because of IE, we also have to check the presence of the constructor
518         //property.
519         if ( !obj || jQuery.type(obj) !== "object" || obj.nodeType || jQuery.
520 isWindow( obj ) ) {
521             return false;
522         }
523         try {
524             //Not own constructor property must be Object
525             if ( obj.constructor &&
526                 !hasOwn.call(obj, "constructor") &&
527                 !hasOwn.call(obj.constructor.prototype, "isPrototypeOf") ) {
528                 return false;
529             }
530         } catch ( e ) {
531             //IE8,9 Will throw exceptions on certain host objects #9897
532             return false;
533         }
534         //Own properties are enumerated firstly, so to speed up,
535         //if last one is own, then all properties are own.
536
537         var key;
538         for ( key in obj ) {}
539
540         return key === undefined || hasOwn.call( obj, key );
```

```
542     },
```

第 519 ~ 521 行：如果参数 obj 满足以下条件之一，则返回 false：

- 参数 obj 可以转换为 false。
- Object.prototype.toString.call(obj) 返回的不是 [object Object]。
- 参数 obj 是 DOM 元素。
- 参数 obj 是 window 对象。

如果参数 obj 不满足以上所有条件，则至少可以确定参数 obj 是对象。

第 523 ~ 533 行：检查对象 obj 是否由构造函数 Object() 创建。如果对象 obj 满足以下所有条件，则认为不是由构造函数 Object() 创建，而是由自定义构造函数创建，返回 false：

- 对象 obj 含有属性 constructor。由构造函数创建的对象都有一个 constructor 属性，默认引用了该对象的构造函数。如果对象 obj 没有属性 constructor，则说明该对象必然是通过对象字面量 {} 创建的。
- 对象 obj 的属性 constructor 是非继承属性。默认情况下，属性 constructor 继承自构造函数的原型对象。如果属性 constructor 是非继承属性，说明该属性已经在自定义构造函数中被覆盖。
- 对象 obj 的原型对象中没有属性 isPrototypeOf。属性 isPrototypeOf 是 Object 原型对象的特有属性，如果对象 obj 的原型对象中没有，说明不是由构造函数 Object() 创建，而是由自定义构造函数创建。
- 执行以上检测时抛出了异常。在 IE 8/9 中，在某些浏览器对象上执行以上检测时会抛出异常，也应该返回 false。

函数 hasOwnProperty() 指向 Object.prototype.hasOwnProperty(property)，用于检查对象是否含有执行名称的非继承属性。

第 539 ~ 541 行：检查对象 obj 的属性是否都是非继承属性。如果没有属性，或者所有属性都是非继承属性，则返回 true。如果含有继承属性，则返回 false。

第 539 行：执行 for-in 循环时，JavaScript 会先枚举非继承属性，再枚举从原型对象继承的属性。

第 541 行：如果对象 obj 的最后一个属性是非继承属性，则认为所有属性都是非继承属性，返回 true；如果最后一个属性是继承属性，即含有继承属性，则返回 false。

6. jQuery.isEmptyObject(object)

方法 jQuery.isEmptyObject(object) 用于检测对象是否是空的（即不包含属性）。例如：

```
jQuery.isEmptyObject( {} )                      // true
jQuery.isEmptyObject( new Object() )            // true
jQuery.isEmptyObject( { foo: "bar" } )           // false
```

方法 jQuery.isEmptyObject(object) 的相关代码如下所示：

```
544     isEmptyObject: function( obj ) {
545         for ( var name in obj ) {
546             return false;
```

```

547         }
548     return true;
549 },

```

第545~548行：for-in循环会同时枚举非继承属性和从原型对象继承的属性，如果有，则立即返回false，否则默认返回true。

2.8.3 解析JSON和XML：jQuery.parseJSON(data)、jQuery.parseXML(data)

1. jQuery.parseJSON(data)

方法jQuery.parseJSON(data)接受一个格式良好的JSON字符串，返回解析后的JavaScript对象。如果传入残缺的JSON字符串可能导致程序抛出异常；如果不传入参数，或者传入空字符串、null、undefined，则返回null。

如果浏览器提供了原生方法JSON.parse()，则使用该方法解析JSON字符串；否则使用(new Function("return"+data))()解析JSON字符串。

方法jQuery.parseJSON(data)的相关代码如下所示：

```

555     parseJSON: function( data ) {
556         if ( typeof data !== "string" || !data ) {
557             return null;
558         }
559
560         // Make sure leading/trailing whitespace is removed (IE can't handle it)
561         data = jQuery.trim( data );
562
563         // Attempt to parse using the native JSON parser first
564         if ( window.JSON && window.JSON.parse ) {
565             return window.JSON.parse( data );
566         }
567
568         // Make sure the incoming data is actual JSON
569         // Logic borrowed from http://json.org/json2.js
570         if ( rvalidchars.test( data.replace( rvalidescape, "@" )
571             .replace( rvalidtokens, "]" )
572             .replace( rvalidbraces, "" ) ) ) {
573
574             return ( new Function( "return " + data ) )();
575
576         }
577         jQuery.error( "Invalid JSON: " + data );
578     },

```

第556~561行：对于非法参数一律返回null。如果参数data不是字符串，或者可以转换为false，则返回null。

第561行：移除开头和末尾的空白符。在IE6/7中，如果不移除就不能正确的解析，例如：

```

typeof ( new Function( 'return ' + '\n{}' ) )();
// 返回 "undefined"

```

第564~566行：尝试使用原生方法JSON.parse()解析JSON字符串，并返回。

JSON 对象含有两个方法：JSON.parse() 和 JSON.stringify()，用于 JSON 字符串和 JavaScript 对象之间的相互转换。下面是它们的语法和使用示例。

JSON.parse() 解析 JSON 字符串为 JSON 对象，其语法如下：

```
JSON.parse(text[, reviver])
//text 待解析为 JSON 对象的字符串
//reviver 可选。在返回解析结果前，对解析结果中的属性值进行修改
```

JSON.parse() 的使用示例如下所示：

```
JSON.parse('{"abc": 123}');
// {"abc": 123}

JSON.parse('{"abc": 123}', function(key, value){
  if(key === '') return value;
  return value * 2;
});
// {"abc": 246}
```

JSON.stringify() 转换 JSON 对象为 JSON 字符串，其语法如下：

```
JSON.stringify(value[, replacer [, space]])
//value 待转换为 JSON 字符串的对象
//replacer 可选。如果 replacer 是函数，转换前先执行 replacer 改变属性值，如果函数
replacer 返回 undefined，则对应的属性不会出现在结果中；如果 replacer 是数组，指定最终字符串中包
含的属性集合，不在数组 replacer 中的属性不会出现在结果中
//space 增加转换后的 JSON 字符串的可读性
```

JSON.stringify() 的使用示例如下所示：

```
JSON.stringify({ a: 1, b: 2 });
// '{"a":1,"b":2}

JSON.stringify({ a: 1, b: 2 }, function(key, value){
  if(key === '') return value;
  if(key === 'a') return value * 10;
  if(key === 'b') return undefined;
  return value;
});
// '{"a":10}

JSON.stringify({ a: 1, b: 2 }, ['b']);
// '{"b":2}

JSON.stringify({ a: 1, b: 2 }, null, 4);
// '\n    "a": 1,\n    "b": 2\n'}
```

JSON 对象、JSON.parse()、JSON.stringify() 在 ECMAScript 5 中被标准化，IE 8 以下的浏览器不支持。关于 JSON 规范和浏览器实现的更多信息请访问以下地址：

<http://json.org/json-zh.html>

<http://www.ecma-international.org/publications/standards/Ecma-262.htm> (ECMAScript 5 第 15.12 节)

https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/JSON

下面回到对方法 `jQuery.parseJSON()` 的分析中来。

第 570 ~ 576 行：在不支持 `JSON.parse()` 的浏览器中，先检查字符串是否合法，如何合法，才会执行 `(new Function("return "+ data))()` 并返回执行结果。检查字符串是否合法的正则和逻辑来自开源 JSON 解析库 `json2.js` (<https://github.com/douglascrockford/JSON-js>)，检测过程分为 4 步，用到了 4 个正则表达式：`rvalidchars`、`rvalidescape`、`rvalidtokens`、`rvalidbraces`，相关代码如下：

```

53     // JSON RegExp
54     rvalidchars = /^[\[],:{}\s]*$/,
55     rvalidescape = /\\"(?:["\\\/bf\nrt]|u[0-9a-fA-F]{4})/g,
56     rvalidtokens = /"[^"\n\r]*"|true|false|null|-?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g,
57     rvalidbraces = /(?:^|:|,) (?:\s*\[]+)/g,
58
59     if ( rvalidchars.test( data.replace( rvalidescape, "@" )
60         .replace( rvalidtokens, "]" )
61         .replace( rvalidbraces, "" ) ) {
62

```

第 54 ~ 57 行：正则 `rvalidescape` 用于匹配转义字符；正则 `rvalidtokens` 用于匹配有效值（字符串、`true`、`false`、`null`、数值）；正则 `rvalidbraces` 用于匹配正确的左方括号 “[”；正则 `rvalidchars` 用于检查字符串是否只含有指定的字符（“[”、“,”、“:”、“{”、“}”、“\s”）。

第 570 ~ 572 行：先利用正则 `rvalidescape` 把转义字符替换为 “@”，为进行下一步替换做准备；再利用正则 `rvalidtokens` 把字符串、`true`、`false`、`null`、数值替换为 “]”；然后利用 `rvalidbraces` 删除正确的左方括号；最后检查剩余字符是否只包含 “[”、“,”、“:”、“{”、“}”、“\s”，如果只包含这些字符，那么认为 JSON 字符串是合法的。

第 574 行：通过构造函数 `Function()` 创建函数对象，然后执行。构造函数 `Function()` 的语法^②如下：

```

new Function( arguments_names..., body)
// 参见 arguments_names...：任意多个字符串参数，每个字符串命名一个或多个要创建的 Function 对象的参数
// 参见 body：一个字符串，指定函数的主体，可以含有任意多条 JavaScript 语句，这些语句之间用分号隔开，可以给该构造函数引用前面的参数设置的任何参数名
// 返回新创建的 Function 对象。调用该函数，将执行 body 指定的 JavaScript 代码

```

第 577 行：如果浏览器不支持 `JSON.parse()`，并且 JSON 字符串不合法，则在最后抛出一个异常。

2. `jQuery.parseXML(data)`

方法 `jQuery.parseXML(data)` 接受一个格式良好的 XML 字符串，返回解析后的 XML 文档。

方法 `jQuery.parseXML()` 使用浏览器原生的 XML 解析函数实现。在 IE 9+ 和其他浏览器中，会使用 `DOMParser` 对象解析；在 IE 9 以下的浏览器中，则使用 `ActiveXObject` 对象解析。相关代码如下所示：

^② 《JavaScript 权威指南 第 5 版 中文版》，核心 JavaScript 参考手册，Function，P640。

```

580     // Cross-browser xml parsing
581     parseXML: function( data ) {
582         var xml, tmp;
583         try {
584             if ( window.DOMParser ) { // Standard
585                 tmp = new DOMParser();
586                 xml = tmp.parseFromString( data , "text/xml" );
587             } else { //IE
588                 xml = new ActiveXObject( "Microsoft.XMLDOM" );
589                 xml.async = "false";
590                 xml.loadXML( data );
591             }
592         } catch( e ) {
593             xml = undefined;
594         }
595         if ( !xml || !xml.documentElement || xml.getElementsByTagName(
"parsererror").length ) {
596             jQuery.error( "Invalid XML: " + data );
597         }
598         return xml;
599     },

```

第 584 ~ 586 行：尝试用标准解析器 DOMParser 解析。DOMParser 在 HTML5 中标准化^②，可以将 XML 或 HTML 字符串解析为一个 DOM 文档。解析时，首先要创建一个 DOMParser 对象，然后使用它的方法 parseFromString() 来解析 XML 或 HTML 字符串。方法 parseFromString() 的语法如下：

```

DOMParser.parseFromString( DOMString str, SupportedType type )
//参数 str: 待解析的 XML 或 HTML 字符串
//参数 type: 支持的类型有"text/html"、"text/xml"、"application/xml"、"application/
xhtml+xml"、"image/svg+xml"
//返回一个解析后的新创建的文档对象

```

在 IE 以外的浏览器中，如果解析失败，方法 parseFromString() 不会抛出任何异常，只会返回一个包含了错误信息的文档对象，如下所示：

```

<parsererror xmlns="http://www.mozilla.org/newlayout/xml/parsererror.xml">
(error description)
<sourcetext>(a snippet of the source XML)</sourcetext>
</parsererror>

```

因此，在第 595 行需要检查解析后的文档中是否包含 <parsererror> 节点，如果包含则表示解析失败，抛出一个更易读的异常。

在 IE 9+ 中，如果解析失败，则会抛出异常。如果抛出异常，在 catch 块中设置 xml 为 undefined，然后抛出一个更易读的异常。

可以运行下面的测试代码来验证上述内容：

^② <http://html5.org/specs/dom-parsing.html#the-domparser-interface>。
<https://developer.mozilla.org/en/DOMParser>。

```
new DOMParser().parseFromString("<a>hello", "text/xml")
```

在Chrome中返回一个包含了错误信息的文档对象：

```
<a>
<parsererror style="display: block; white-space: pre; border: 2px solid #c77;
padding: 0 1em 0 1em; margin: 1em; background-color: #fdd; color: black">
<h3>This page contains the following errors:</h3>
<div style="font-family:monospace;font-size:12px">
    error on line 1 at column 9: Extra content at the end of the document
</div>
<h3>Below is a rendering of the page up to the first error.</h3>
</parsererror>hello
</a>
```

在IE9中则抛出异常：

```
"DOM Exception: SYNTAX_ERR (12)"
XML5602: 输入意外结束。
, 行 1 字符 9
```

下面回到对方法jQuery.parseXML()源码的分析中来。

第588~594行：IE9以下的浏览器不支持DOMParser，需要使用微软的XML解析器Microsoft.XMLDOM解析。解析步骤依次为：创建一个空的XML文档对象，设该文档对象为同步加载，调用方法loadXML()解析XML字符串。

如果解析成功，方法loadXML()会返回true，解析结果存放在创建的XML文档对象中；如果解析失败，方法loadXML()会返回false（不会抛出异常），并设置文档根节点documentElement为null。

第595~597行：如果解析失败，则抛出一个更易读的异常。如果满足以下条件之一，则认为解析失败：

- 在IE9+中，通过标准XML解析器DOMParser解析失败，此时!xml为true。
- 在IE9以下的浏览器中，通过微软的XML解析器Microsoft.XMLDOM解析失败，此时!xml.documentElement为true。
- 在其他浏览器中，通过标准XML解析器DOMParser解析失败，此时xml.getElementsByTagName("parsererror").length可以转换为true。

第598行：如果解析成功，则返回解析结果。

下面是一些扩展阅读：

XML文档属性和方法：

[http://msdn.microsoft.com/en-us/library/windows/desktop/ms763798\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms763798(v=vs.85).aspx)

[http://msdn.microsoft.com/en-us/library/windows/desktop/ms757828\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms757828(v=vs.85).aspx)

解析XML文档：

http://www.w3schools.com/xml/xml_parser.asp

HTML5规范中的XML序列化：

<http://html5.org/specs/dom-parsing.html#xmlserializer>

2.8.4 jQuery.globalEval(code)

方法 `jQuery.globalEval(code)` 用于在全局作用域中执行 JavaScript 代码。很多时候我们希望 JavaScript 代码是在全局作用域中执行，例如，当动态加载并执行 JavaScript 代码时。

在 IE 中，可以调用方法 `execScript()` 让 JavaScript 代码在全局作用域中执行；在其他浏览器中，则需要在一个自调用匿名函数中调用 `eval()` 执行 JavaScript 代码，自调用匿名函数确保了执行环境是全局作用域。相关代码如下所示：

```

603      // Evaluates a script in a global context
604      // Workarounds based on findings by Jim Driscoll
605      // http://weblogs.java.net/blog/driscoll/archive/2009/09/08/eval-javascript-
global-context
606      globalEval: function( data ) {
607          if ( data && rnotwhite.test( data ) ) {
608              // We use execScript on Internet Explorer
609              // We use an anonymous function so that context is window
610              // rather than jQuery in Firefox
611              ( window.execScript || function( data ) {
612                  window[ "eval" ].call( window, data );
613              } )( data );
614      }
615  },

```

方法 `execScript()` 在全局作用域中按照指定的脚本语言执行脚本代码，默认语言是 Jscript，没有返回值。该方法的语法[⊖]如下：

```

execScript( code, language )
// 参数 code: 待执行的脚本代码
// 参数 language: 脚本语言, 可选值有 JavaScript、JavaScript1.1、JavaScript1.2、JavaScript1.3、
Jscript、VBS、VBScript, 默认是 Jscript

```

Chrome 的早期版本曾支持方法 `execScript()`，现已不支持。

方法 `eval()` 在调用它的作用域中计算或执行 JavaScript 代码。如果 JavaScript 代码是一条表达式，则计算并返回计算结果；如果含有一条或多条 JavaScript 语句，则执行这些语句，如果最后一条语句有返回值，则返回这个值，否则返回 `undefined`。该方法的语法如下：

```

eval( code )
// 参数 code: 待执行的 JavaScript 表达式或语句

```

第 611 ~ 613 行：如果支持方法 `execScript()`，则执行 `execScript(data)`；否则创建一个自调用匿名函数，在函数内部执行 “`window["eval"].call(window, data);`”，读者可以访问源码注释第 605 行的网页地址查看如此书写的原因。

2.8.5 jQuery.camelCase(string)

方法 `jQuery.camelCase(string)` 转换连字符式的字符串为驼峰式，用于 CSS 模块和数据缓存模块。例如：

[⊖] [http://msdn.microsoft.com/en-us/library/ie/ms536420\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ie/ms536420(v=vs.85).aspx)。

```
jquery.camelCase( 'background-color' );
// "backgroundColor"
```

方法 `jQuery.camelCase(string)` 的相关代码如下所示：

```
65    // Matches dashed string for camelizing
66    rdashAlpha = /-([a-z]|[0-9])/ig,
67    rmsPrefix = /^-ms-/,
68
69    // Used by jQuery.camelCase as callback to replace()
70    fcamelCase = function( all, letter ) {
71        return ( letter + "" ).toUpperCase();
72    },
73
74    // Convert dashed to camelCase; used by the css and data modules
75    // Microsoft forgot to hump their vendor prefix (#9572)
76    camelCase: function( string ) {
77        return string.replace( rmsPrefix, "ms-" ).replace( rdashAlpha, fcamelCase );
78    }
79
```

第 66 行：正则 `rdashAlpha` 用于匹配字符串中连字符“-”和其后的第一个字母或数字。如果连字符“-”后是字母，则匹配部分会被替换为对应的大写字母；如果连字符“-”后是数字，则会删掉连字符“-”，保留数字。

第 67 行：正则 `rmsPrefix` 用于匹配字符串中前缀“-ms-”，匹配部分会被替换为“ms-”。这么做是因为在 IE 中，连字符式的样式名前缀“-ms-”对应小写的“ms”，而不是驼峰式的“Ms”。例如，“-ms-transform”对应“msTransform”而不是“MsTransform”。在 IE 以外的浏览器中，连字符式的样式名则可以正确地转换为驼峰式，例如，“-moz-transform”对应“MozTransform”。

第 70 ~ 72 行：函数 `fcamelCase()` 负责把连字符后的字母转换为大写并返回。

第 619 ~ 621 行：在方法 `jQuery.camelCase()` 中，先用正则 `rmsPrefix` 匹配前缀“-ms-”，如果有则修正为“ms-”；然后用正则 `rdashAlpha` 匹配连字符“-”和其后的第一个字母或数字，并用字符串方法 `replace()` 和函数 `fcamelCase()` 把匹配部分替换为对应的大写字母或数字。

2.8.6 `jQuery.nodeName(elem, name)`

方法 `jQuery.nodeName(elem, name)` 用于检查 DOM 元素的节点名称（即属性 `nodeName`）与指定的值是否相等，检查时忽略大小写。

DOM 元素的属性 `nodeName` 返回该元素的节点名称；对于 HTML 文档，始终返回其大写形式；对于 XML 文档，因为 XML 文档区分大小写，所以返回值与源代码中的形式一致。在方法 `jQuery.nodeName(elem, name)` 中会把属性 `nodeName` 和参数 `name` 转换为大写形式后再做比较，即忽略大小写。相关代码如下所示：

```
623    nodeName: function( elem, name ) {
624        return elem.nodeName && elem.nodeName.toUpperCase() === name.toUpperCase();
625    },
626
```

第 624 行：把属性 elem.nodeName 和参数 name 都转换为大写再做比较。在执行 elem.nodeName.toUpperCase() 前先检查 elem.nodeName 是否存在，可以有效地避免参数 elem 不是 DOM 元素，或者参数 elem 没有属性 nodeName 导致的错误。

2.8.7 jQuery.trim(str)

方法 jQuery.trim(str) 用于移除字符串开头和结尾的空白符。如果传入的参数是 null 或 undefined，则返回空字符串；如果传入的参数是对象，则先获取对象的字符串表示，然后移除开头和结尾的空白符，并返回。相关代码如下所示：

```

43     // Check if a string has a non-whitespace character in it
44     rnotwhite = /\S/,
45
46     // Used for trimming whitespace
47     trimLeft = /^[\s]+/,
48     trimRight = /[\s]+\$/,
49
50     trim = String.prototype.trim,
51
52     // IE doesn't match non-breaking spaces with \s
53     if ( rnotwhite.test( "\xA0" ) ) {
54         trimLeft = /^[\s\xA0]+/;
55         trimRight = /[\s\xA0]+\$/;
56     }
57
58     // Use native String.trim function wherever possible
59     trim: trim ?
60         function( text ) {
61             return text == null ?
62                 "" :
63                 trim.call( text );
64         } :
65
66         // Otherwise use our own trimming functionality
67         function( text ) {
68             return text == null ?
69                 "" :
70                 text.toString().replace( trimLeft, "" ).replace( trimRight, "" );
71     },
72
73     trim
74 }
```

第 47 ~ 48 行：正则 trimLeft 用于匹配字符串开头的空白符；trimRight 用于匹配字符串结尾的空白符。

第 911 ~ 914 行：在 IE 9 以下的浏览器中，\s 不匹配不间断空格 \xA0，需要为正则 trimLeft 和 trimRight 加上 “\xA0”。

第 669 ~ 681 行：如果浏览器支持 String.prototype.trim() 则“借鸡生蛋”，String.prototype.trim() 是 ECMAScript 5 新增的 String 原型方法；如果不支持，则先调用方法 toString() 得到参数 text 的字符串表示，然后调用方法 replace() 把正则 trimLeft 和 trimRight 匹配到的空白符替换为空字符串。如果参数是 null 或 undefined，则返回空字符串。

2.8.8 数组操作方法: `jQuery.makeArray(obj)`、`jQuery.inArray(value, array [, fromIndex])`、`jQuery.merge(first, second)`、`jQuery.grep(array, function(elementOfArray, indexInArray) [, invert])`

1. `jQuery.makeArray(obj)`

方法 `jQuery.makeArray(obj)` 可以将一个类数组对象转换为真正的数组。

在 jQuery 内部，还可以为方法 `jQuery.makeArray()` 传入第二个参数，这样，第一个参数中的元素将被合并入第二个参数，最后会返回第二个参数，此时返回值的类型不一定是真正的数组。

方法 `jQuery.makeArray(obj)` 的源码如下：

```

89      push = Array.prototype.push,
683      // results is for internal usage only
684      makeArray: function( array, results ) {
685          var ret = results || [];
686
687          if ( array != null ) {
688              // The window, strings (and functions) also have 'length'
689              // Theaked logic slightly to handle Blackberry 4.7 RegExp issues #6930
690              var type = jQuery.type( array );
691
692              if ( array.length == null
693                  || type === "string"
694                  || type === "function"
695                  || type === "regexp"
696                  || jQuery.isWindow( array ) ) {
697                  push.call( ret, array );
698              } else {
699                  jQuery.merge( ret, array );
700              }
701          }
702
703          return ret;
704      },

```

第 684 行：定义方法 `jQuery.makeArray(array, results)`，它接受 2 个参数：

- 参数 `array`: 待转换对象，可以是任何类型。
- 参数 `results`: 仅在 jQuery 内部使用。如果传入参数 `results`，则在该参数上添加元素。

第 685 行：定义返回值 `ret`。如果传入了参数 `results` 则把该参数作为返回值，否则新建一个空数组作为返回值。

第 687 行：过滤参数 `array` 是 `null`、`undefined` 的情况。

第 690 ~ 693 行：如果参数 `array` 满足以下条件之一，则认为该参数不是数组，也不是类数组对象，调用数组方法 `push()` 把该参数插入返回值 `ret` 的末尾：

- 参数 `array` 没有属性 `length`。
- 参数 `array` 是字符串，属性 `length` 返回字符串中的字符个数。
- 参数 `array` 是函数，属性 `length` 返回函数声明时的参数个数。

□ 参数 array 是 window 对象，属性 length 返回窗口中的框架（frame、iframe）个数。

□ 参数 array 是正则对象，在 Blackberry（黑莓）4.7 中，正则对象也有 length 属性。

注意，第 693 行插入元素时执行的是 push.call(ret, array)，而不是 ret.push(array)，这是因为返回值 ret 不一定是真正的数组。如果只传入参数 array，则返回值 ret 是真正的数组；如果还传入了第二个参数 result，则返回值 ret 的类型取决于该参数的类型。

第 694 ~ 696 行：否则认为参数 array 是数组或类数组对象，调用方法 jQuery.merge() 把该参数合并到返回值 ret 中。

第 699 行：最后返回 ret。

2. jQuery.inArray(value, array[, fromIndex])

方法 jQuery.inArray(value, array[, fromIndex]) 在数组中查找指定的元素并返回其下标，未找到则返回 -1。相关代码如下所示：

```

702     inArray: function( elem, array, i ) {
703         var len;
704
705         if ( array ) {
706             if ( indexOf ) {
707                 return indexOf.call( array, elem, i );
708             }
709
710             len = array.length;
711             i = i ? i < 0 ? Math.max( 0, len + i ) : i : 0;
712
713             for ( ; i < len; i++ ) {
714                 // Skip accessing in sparse arrays
715                 if ( i in array && array[ i ] === elem ) {
716                     return i;
717                 }
718             }
719         }
720
721         return -1;
722     },

```

第 702 行：定义方法 jQuery.inArray(elem, array, i)，它接受 3 个参数：

□ 参数 elem：要查找的值。

□ 参数 array：数组，将遍历这个数组来查找参数 value 在其中的下标。

□ 参数 i：指定开始查找的位置，默认是 0 即查找整个数组。

第 705 行：过滤 array 可以转换为 false 的情况。

第 706 ~ 708 行：如果浏览器支持数组方法 indexOf()，则调用它并返回下标。该方法在 ECMAScript 5 中被标准化。

第 711 行：修正参数 i。如果未指定参数 i，则初始化为 0，表示默认从头开始遍历；如果 i 小于 0，则加上数组长度 len，即从数组末尾开始计算，例如，-1 表示最后一个元素，-2 表示倒数第二个元素，以此类推。注意，这里调用了 Math.max() 方法在 0 和 len+i 之间取最大值，即如果 len+i 依然小于 0，则把 i 修正为 0，仍然从头开始遍历。

第 713 ~ 718 行：从指定位置开始遍历数组，查找与指定值 elem 相等的元素，并返回

其下标。先检测 `i in array`，如果结果是 `false`，说明数组 `array` 的下标是不连续的，也不需要与指定值 `elem` 比较；然后检测 `array[i] === elem`，如果结果是 `true`，则返回当前下标。这里使用等同运算符（`==`）来避免类型转换。

第 721 行：如果没有找到与指定值相等的元素，则默认返回 `-1`。从方法 `jQuery.inArray()` 的命名来看，这个方法应该返回 `true` 或 `false`，而它实际上返回的却是下标，因此把方法名改为 `indexOf` 或许更合适些，但是这个方法从 `jQuery 1.2` 就一直存在，如果修改则会导致严重的向后兼容问题，所以返回值和方法名都不宜修改。

通常我们会比较 `jQuery.inArray()` 的返回值是否大于 `0`，来判断某个指定的元素是否是数组中的元素，就像下面这样：

```
if( jQuery.inArray( elem, array ) > 0 ){
    //elem 是 array 中的元素
}
```

上面的写法比较繁琐，特别是当 `if` 语句检测的条件是复合布尔表达式时，可读性会很差；可以用按位非运算符（`~`）简化上面的代码：

```
if( ~jQuery.inArray( elem, array ) ){
    //elem 是 array 中的元素
}
```

按位非运算符（`~`）会将运算数的所有位取反，相当于改变它的符号并且减 1，例如：

```
~-1 == 0; //true
~0 == -1; //true
~1 == -2; //true
~2 == -3; //true
```

更进一步，可以结合使用按位非运算符（`~`）和逻辑非运算符（`!`）把 `jQuery.inArray()` 的返回值转换为布尔型：

```
!~jQuery.inArray( elem, array )
//如果 elem 可以匹配 array 中的某个元素，则该表达式的值为 true
//如果 elem 匹配不到 array 中的元素，则该表达式的值为 false
```

3. `jQuery.merge(first, second)`

方法 `jQuery.merge(first, second)` 用于合并两个数组的元素到第一个数组中。事实上，第一个参数可以是数组或类数组对象，即必须含有整型（或可以转换为整型）属性 `length`；第二个参数则可以是数组、类数组对象或任何含有连续整型属性的对象。

方法 `jQuery.merge()` 的合并行为是破坏性的，将第二个数组中的元素添加到第一个数组中后，第一个数组就被改变了。如果希望原来的第一个数组不被改变，可以在调用 `jQuery.merge()` 之前创建一份第一个数组的副本：

```
var newArray = $.merge([], oldArray);
```

方法 `jQuery.merge(first, second)` 的相关代码如下所示：

```
724     merge: function( first, second ) {
```

```

725     var i = first.length,
726         j = 0;
727
728     if ( typeof second.length === "number" ) {
729         for ( var l = second.length; j < l; j++ ) {
730             first[ i++ ] = second[ j ];
731         }
732
733     } else {
734         while ( second[j] !== undefined ) {
735             first[ i++ ] = second[ j++ ];
736         }
737     }
738
739     first.length = i;
740
741     return first;
742 },

```

第 724 行：定义方法 `jQuery.merge(first, second)`，它接受 2 个参数：

- 参数 `first`：数组或类数组对象，必须含有整型（或可以转换为整型）属性 `length`，第二个数组 `second` 中的元素会被合并到该参数中。
- 参数 `second`：数组、类数组对象或任何含有连续整型属性的对象，其中的元素会被合并到第一个参数 `first` 中。

第 725 行：初始化变量 `i` 为 `first.length`，该变量指示了插入新元素时的下标。`first.length` 必须是整型或可以转换为整型，否则后面执行 `i++` 时会返回 `NaN`。

第 728 ~ 731 行：如果参数 `second` 的属性 `length` 是数值类型，则把该参数当作数组处理，把其中的所有元素都添加到参数 `first` 中。

第 733 ~ 737 行：如果参数 `second` 没有属性 `length`，或者属性 `length` 不是数值类型，则把该参数当作含有连续整型（或可以转换为整型）属性的对象，例如，`{ 0:'a', 1:'b' }`，把其中的非 `undefined` 元素逐个插入参数 `first` 中。

第 739 行：修正 `first.length`。因为参数 `first` 可能不是真正的数组，所以需要手动维护属性 `length` 的值。

第 741 行：返回改变后的参数 `first`。

4. `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])`

方法 `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 用于查找数组中满足过滤函数的元素，原数组不会受影响。

如果参数 `invert` 未传入或是 `false`，元素只有在过滤函数返回 `true`，或者返回值可以转换为 `true` 时，才会被保存在最终的结果数组中，即返回一个满足回调函数的元素数组；如果参数 `invert` 是 `true`，则情况正好相反，返回的是一个不满足回调函数的元素数组。

该方法的相关代码如下所示：

```

744     grep: function( elems, callback, inv ) {
745         var ret = [], retVal;
746         inv = !!inv;

```

```

747
748      // Go through the array, only saving the items
749      // that pass the validator function
750      for ( var i = 0, length = elems.length; i < length; i++ ) {
751          retVal = !callback( elems[ i ], i );
752          if ( inv !== retVal ) {
753              ret.push( elems[ i ] );
754          }
755      }
756
757      return ret;
758  },

```

第744行：定义方法jQuery.grep(elems, callback, inv)，它接受3个参数：

- 参数array：待遍历查找的数组。
- 参数callback：过滤每个元素的函数，执行时被传入两个参数：当前元素和它的下标。该函数应该返回一个布尔值。
- 参数inv：如果参数inv是false或未传入，方法jQuery.grep()会返回一个满足回调函数的元素数组；如果参数inv是true，则返回一个不满足回调函数的元素数组。

第746行：遍历数组elems，为每个元素执行过滤函数。如果参数inv为true，把执行结果为false的元素放入结果数组ret；如果inv为false，则把执行结果为true的元素放入结果数组ret。

第757行：最后返回结果数组ret。

2.8.9 jQuery.guid、jQuery.proxy(function, context)

1. jQuery.guid

属性jQuery.guid是一个全局计数器，用于jQuery事件模块和缓存模块。在jQuery事件模块中，每个事件监听函数会被设置一个guid属性，用来唯一标识这个函数；在缓存模块中，通过在DOM元素上附加一个唯一标识，来关联该元素和该元素对应的缓存。属性jQuery.guid初始值为1，使用时自增1，相关代码如下所示：

```

793      // A global GUID counter for objects
794      guid: 1,
795
// jQuery.data( elem, name, data, pvt /* Internal Use Only */ )
1679                  elem[ internalKey ] = id = ++jQuery.uuid;
1680
// jQuery.event.add: function( elem, types, handler, data, selector )
2861                  handler.guid = jQuery.guid++;

```

2. jQuery.proxy(function, context)

方法jQuery.proxy(function, context)接受一个函数，返回一个新函数，新函数总是持有特定的上下文。这个方法有两种用法：

(1) jQuery.proxy(function, context)

参数function是将被改变上下文的函数，参数context是上下文。指定参数function的上下文始终为参数content。

(2) `jQuery.proxy(context, name)`

参数 `name` 是参数 `context` 的属性。指定参数 `name` 对应的函数的上下文始终为参数 `context`。该方法的相关代码如下所示：

```

796     // Bind a function to a context, optionally partially applying any
797     // arguments.
798     proxy: function( fn, context ) {
799         if ( typeof context === "string" ) {
800             var tmp = fn[ context ];
801             context = fn;
802             fn = tmp;
803         }
804
805         // Quick check to determine if target is callable, in the spec
806         // this throws a TypeError, but we will just return undefined.
807         if ( !jQueryisFunction( fn ) ) {
808             return undefined;
809         }
810
811         // Simulated bind
812         var args = slice.call( arguments, 2 ),
813             proxy = function() {
814                 return fn.apply( context, args.concat( slice.call( arguments ) ) );
815             };
816
817         // Set the guid of unique handler to the same of original handler, so
it can be removed
818         proxy.guid = fn.guid = fn.guid || proxy.guid || jQuery.guid++;
819
820         return proxy;
821     },

```

第 798 行：定义方法 `jQuery.proxy(fn, context)`，参数有两种格式：

- `jQuery.proxy(fn, context)`
- `jQuery.proxy(context, name)`

第 799 行：修正参数 `fn` 和 `context`。如果第二个参数是字符串，说明参数格式是 `jQuery.proxy(context, name)`，修正为 `jQuery.proxy(fn, context)`。

第 807 ~ 809 行：如果参数 `fn` 不是函数，则返回 `undefined`。

第 812 行：收集多余参数。如果调用 `jQuery.proxy()` 时，除了传入参数 `fn`、`context` 之外，还传入了其他参数，那么在调用函数 `fn` 时，这些多余的参数将会优先传入。这里借用数组方法 `slice()` 来获取参数对象 `arguments` 中 `fn`、`context` 后的其他参数。下面的例子测试了传入多余参数的情况：

```

var proxied = $.proxy( function(){
    console.log( this );           // Object
    console.log( arguments );      // [1, 2, 3]
}, {}, 1, 2, 3 );
proxied(4, 5);
// 在控制台依次打印：
// Object
// [1, 2, 3, 4, 5]

```

第813~815行：创建一个代理函数，在代理函数中调用原始函数fn，调用时通过方法apply()指定上下文。代理函数通过闭包机制引用context、args、slice。

第818行：为代理函数设置与原始函数相同的唯一标识guid。如果原始函数没有，则重新分配一个。

相同的唯一标识将代理函数和原始函数关联了起来。例如，在jQuery事件系统中，如果为DOM元素绑定了事件监听函数的代理函数，当移除事件时，即使传入的是原始函数，jQuery也能通过唯一标识guid移除正确的函数。

第820行：最后返回创建的代理函数。

2.8.10 jQuery.access(elems, key, value, exec, fn(elem, key, value), pass)

方法jQuery.access(elems, key, value, exec, fn(elem, key, value), pass)可以为集合中的元素设置一个或多个属性值，或者读取第一个元素的属性值。如果设置的属性值是函数，并且参数exec是true时，还会执行函数并取其返回值作为属性值。

方法jQuery.access()为.attr()、.prop()、.css()提供支持，这三个方法在调用jQuery.access()时，参数exec为true，参数fn是同时支持读取和设置属性的函数（例jQuery.attr()、jQuery.prop()），相关代码如下所示：

```
// .attr()
2166     attr: function( name, value ) {
2167         return jQuery.access( this, name, value, true, jQuery.attr );
2168     },
2169
// .prop()
2170     prop: function( name, value ) {
2171         return jQuery.access( this, name, value, true, jQuery.prop );
2172     },
2173
// .css()
6460     jQuery.fn.css = function( name, value ) {
6461         ...
6462         return jQuery.access( this, name, value, true, function( elem, name, value ) {
6463             return value !== undefined ?
6464                 jQuery.style( elem, name, value ) :
6465                 jQuery.css( elem, name );
6466         });
6467     };
6468 
```

方法jQuery.access(elems, key, value, exec, fn, pass)的相关代码如下所示：

```
823     // Mutifunctional method to get and set values to a collection
824     // The value/s can optionally be executed if it's a function
825     access: function( elems, key, value, exec, fn, pass ) {
826         var length = elems.length;
827
828         // Setting many attributes
829         if ( typeof key === "object" ) {
830             for ( var k in key ) {
831                 jQuery.access( elems, k, key[k], exec, fn, value );
832             }
833         } else {
834             jQuery.access( elems, key, value, exec, fn, pass );
835         }
836     },
837
838     // Getting many values
839     slice: function( elems, start, end ) {
840         var ret = [];
841         for ( ; start < end; start++ ) {
842             ret.push( jQuery.access( elems, start ) );
843         }
844         return ret;
845     }
846 
```

```

832         }
833     return elems;
834 }
835
836 // Setting one attribute
837 if ( value !== undefined ) {
838     // Optionally, function values get executed if exec is true
839     exec = !pass && exec && jQuery.isFunction(value);
840
841     for ( var i = 0; i < length; i++ ) {
842         fn( elems[i], key, exec ? value.call( elems[i], i, fn( elems[i],
843             key ) ) : value, pass );
844     }
845
846     return elems;
847 }
848
849 // Getting an attribute
850 return length ? fn( elems[0], key ) : undefined;
851 },

```

第 825 行：定义方法 `jQuery.access(elems, key, value, exec, fn, pass)`，它接受 6 个参数：

- 参数 `elems`: 元素集合，通常是 `jQuery` 对象。
- 参数 `key`: 属性名或含有键值对的对象。
- 参数 `value`: 属性值或函数。当参数 `key` 是对象时，该参数为 `undefined`。
- 参数 `exec`: 布尔值，当属性值是函数时，该参数指示了是否执行函数。
- 参数 `fn`: 回调函数，同时支持读取和设置属性。
- 参数 `pass`: 布尔值，该参数在功能上与参数 `exec` 重叠，并且用法相当繁琐，可以忽略这个参数。

第 829 ~ 834 行：如果参数 `key` 是对象，表示要设置多个属性，则遍历参数 `key`，为每个属性递归调用方法 `jQuery.access()`，遍历完后返回元素集合 `elems`。

第 837 ~ 846 行：如果参数 `value` 不是 `undefined`，表示要设置单个属性，则遍历元素集合 `elems`，为每个元素调用回调函数 `fn`，遍历完后返回元素集合 `elems`。如果参数 `exec` 为 `true`，并且参数 `value` 是函数，则执行参数 `value`，并取其返回值作为属性值。

第 849 行：读取一个属性。如果元素集合 `elems` 不为空，则为第一个元素调用回调函数 `fn`，读取参数 `key` 对应的属性值；否则返回 `undefined`。

2.8.11 `jQuery.error(message)`、`jQuery.noop()`、`jQuery.now()`

方法 `jQuery.error(message)` 接受一个字符串，抛出一个包含了该字符串的异常。开发插件时可以覆盖这个方法，用来显示更有用或更多的错误提示消息。

方法 `jQuery.noop()` 表示一个空函数。当希望传递一个什么也不做的函数时，可以使用这个空函数。开发插件时，这个方法可以作为可选回调函数的默认值，如果没有提供回调函数，则执行 `jQuery.noop()`。

方法 `jQuery.now()` 返回当前时间的毫秒表示，是 `(newDate()).getTime()` 的简写。

上面3个方法的相关代码如下所示：

```

551     error: function( msg ) {
552         throw new Error( msg );
553     },
561     noop: function() {},
582     now: function() {
583         return ( new Date() ).getTime();
584     },

```

2.8.12 浏览器嗅探：jQuery.uaMatch(ua)、jQuery.browser

属性 `jQuery.browser` 提供了访问当前页面的浏览器的信息，其中包含最流行的4种浏览器类型（IE、Mozilla、Webkit、Opera）和版本信息，属性值的结构如下：

```

// jQuery.browser
{
    webkit/opera/msie/mozilla: true,
    version: '版本号',
}

```

Chrome 和 Safari 使用 Webkit 作为内核引擎，因此如果 `jQuery.browser.webkit` 为 `true` 则表示浏览器是 Chrome 或 Safari；如果 `jQuery.browser.mozilla` 为 `true`，则表示浏览器是 Mozilla Firefox。

`jQuery.browser` 通过解析 `navigator.userAgent` 来获取浏览器类型和版本号，这种技术也称为浏览器嗅探技术，用于解决浏览器不兼容问题；`navigator` 是全局对象 `window` 的属性，指向一个 `Navigator` 对象，包含了正在使用的浏览器的信息；`navigator.userAgent` 包含了浏览器用于 HTTP 请求的用户代理头（User-Agent）的值。

应该避免编写基于特定浏览器类型或版本号的代码，因为这会导致代码与特定的浏览器类型或版本紧密绑定在一起，另外，用户或浏览器也可以修改 `navigator.userAgent`，欺骗脚本和服务端；解决浏览器不兼容问题的更好做法是基于浏览器功能测试编写代码，具体请参阅第7章。

对 `navigator.userAgent` 的解析由方法 `jQuery.uaMatch(ua)` 实现，相关代码如下所示：

```

59     // Useragent RegExp
60     rwebkit = /(webkit)[ \/]([\w.]+)/,
61     ropera = /(opera)(?:.*version)?[ \/]([\w.]+)/,
62     rmsie = /(msie) ([\w.]+)/,
63     rmozilla = /(mozilla)(?:.*? rv:([\w.]+))?/,
64
65     // Keep a userAgent string for use with jQuery.browser
66     userAgent = navigator.userAgent,
67
68     // Use of jQuery.browser is frowned upon.
69     // More details: http://docs.jquery.com/Utilities/jQuery.browser
70     uaMatch: function( ua ) {
71         ua = ua.toLowerCase();
72     }

```

```

861     var match = rwebkit.exec( ua ) ||
862       ropera.exec( ua ) ||
863       rmsie.exec( ua ) ||
864       ua.indexOf("compatible") < 0 && rmozilla.exec( ua ) ||
865       [];
866
867     return { browser: match[1] || "", version: match[2] || "0" };
868   },
869
891   browser: {}
892 });
893
894 browserMatch = jQuery.uaMatch( userAgent );
895 if ( browserMatch.browser ) {
896   jQuery.browser[ browserMatch.browser ] = true;
897   jQuery.browser.version = browserMatch.version;
898 }
899
900 // Dprecated, use jQuery.browser.webkit instead
901 if ( jQuery.browser.webkit ) {
902   jQuery.browser.safari = true;
903 }

```

第 60 ~ 63 行：定义用于解析用户代理 `navigator.userAgent` 的 4 个正则表达式：`rwebkit`、`ropera`、`rmsie`、`rmozilla`。每个正则包含两个分组：浏览器类型特征字符串和浏览器版本特征字符串。

第 858 ~ 868 行：定义方法 `jQuery.uaMatch(ua)`，用于解析当前浏览器的类型和版本号。在方法 `jQuery.uaMatch(ua)` 中，依次尝试用 4 个正则表达式匹配用户代理 `navigator.userAgent`，并返回一个包含了匹配结果的对象，对象的结构是：

```
{
  browser: 分组 1 或空字符串 ,
  version: 分组 2 或字符串 "0"
}
```

第 899 ~ 903 行：调用方法 `jQuery.uaMatch(ua)` 解析用户代理 `navigator.userAgent`，并把解析结果重新封装为 `jQuery.browser`。

2.8.13 小结

构造 `jQuery` 对象模块的静态属性和方法总结如图 2-9 所示。

2.9 总结

在本章中，对构造 `jQuery` 对象模块做了完整的分析。首先介绍了 `jQuery` 构造函数的 7 种用法（见图 2-1）；其次从整体上总结和分析了源码的总体结构（见代码清单 2-1），并对难点专门做了解答；然后分析了构造函数 `jQuery.fn.init()` 的 12 个分支（见图 2-4），以及转换复杂 HTML 代码为 DOM 元素的方法 `jQuery.buildFragment()`（见图 2-5）和 `jQuery.clean()`（见图 2-7）；接着分析了核心工具方法 `jQuery.extend()` 和 `jQuery.fn.extend()`；最后分析了其他的原型属性和方法（见图 2-8）、静态属性和方法（见图 2-9）。

jQuery 静态属性和方法	
<code>jQuery.noConflict([removeAll])</code>	释放 jQuery 对全局变量 \$、jQuery 的控制权
<code>jQuery.isReady</code>	
<code>jQuery.readyWait</code>	
<code>jQuery.holdReady()</code>	
<code>jQuery.ready()</code>	
<code>jQuery.bindReady()</code>	
<code>jQueryisFunction(obj)</code>	判断传入的参数是否是函数
<code>jQuery.isArray(obj)</code>	判断传入的参数是否是数组
<code>jQuery.isWindow(obj)</code>	判断传入的参数是否是 window 对象
<code>jQuery.isNumeric(value)</code>	判断传入的参数是否是数字，或者看起来是否像数字
<code>jQuery.type(obj)</code>	判断参数的内建 JavaScript 类型
<code>jQuery.isPlainObject(object)</code>	判断传入的参数是否是“纯粹”的对象
<code>jQuery.isEmptyObject(object)</code>	检测对象是否是空的（即不包含属性）
<code>jQuery.parseJSON(json)</code>	接受一个格式良好的 JSON 字符串， 返回解析后的 JavaScript 对象
<code>jQuery.parseXML(data)</code>	接受一个格式良好的 XML 字符串， 返回解析后的 XML 文档
<code>jQuery.globalEval(code)</code>	在全局作用域中执行 JavaScript 代码
<code>jQuery.camelCase(string)</code>	转换连字符式的字符串为驼峰式
<code>jQuery.nodeName(elem, name)</code>	检查 DOM 元素的节点名称（即属性 nodeName） 与指定的值是否相等，检查时忽略大小写
<code>jQuery.trim(str)</code>	移除字符串开头和结尾的空白符
<code>jQuery.each(collection, callback(indexInArray, valueOfElement))</code>	通用的遍历迭代方法，用于无缝地遍历对象和数组
<code>jQuery.makeArray(obj)</code>	将一个类数组对象转换为真正的数组
<code>jQuery.inArray(value, array[, fromIndex])</code>	在数组中查找指定的元素并返回其下标，未找到则返回 -1
<code>jQuery.merge(first, second)</code>	合并两个数组的元素到第一个数组中
<code>jQuery.grep(array, function(elementOfArray, indexInArray) [, invert])</code>	查找数组中满足过滤函数的元素，原数组不会受影响
<code>jQuery.map(arrayOrObject, callback(value, indexOrKey))</code>	对数组中的每个元素或对象的每个属性调用一个回调函数， 并将回调函数的返回值放入一个新的数组中
<code>jQuery.guid</code>	全局计数器
<code>jQuery.proxy(function, context)</code>	接受一个函数，返回一个新函数，新函数总是持有特定的上下文
<code>jQuery.error(msg)</code>	接收一个字符串，抛出一个包含了该字符串的异常
<code>jQuery.noop()</code>	一个空函数
<code>jQuery.now()</code>	返回当前时间的毫秒表示
<code>jQuery.access()</code>	为集合中的元素设置一个或多个属性值，或者读取第一个元素的属性值
<code>jQuery.usMatch(ua)</code>	解析当前浏览器的类型和版本号
<code>jQuery.browser</code>	含有当前浏览器的类型和版本号
<code>jQuery.sub()</code>	创建一个新的 jQuery 副本

图 2-9 构造 jQuery 对象模块的静态属性和方法

第三部分

底层支持模块

- 第 3 章 选择器 Sizzle
- 第 4 章 异步队列 Deferred Object
- 第 5 章 数据缓存 Data
- 第 6 章 队列 Queue
- 第 7 章 浏览器功能测试 Support



第 3 章

选择器 Sizzle

Sizzle 是一款纯 JavaScript 实现的 CSS 选择器引擎，它具有以下特性：

- 完全独立，无库依赖。
- 相较于大多数常用选择器其性能非常有竞争力。
- 压缩和开启 gzip 后只有 4 KB。
- 具有高扩展性和易于使用的 API。
- 支持多种浏览器，如 IE 6.0+、Firefox 3.0+、Chrome 5+、Safari 3+、Opera 9+。

W3C Selectors API[⊖]规范定义了方法 querySelector() 和 querySelectorAll()，它们用于根据 CSS 选择器规范[⊖]定位文档中的元素，但是老版本的浏览器（如 IE 6、IE 7）不支持这两个方法。在 Sizzle 内部，如果浏览器支持方法 querySelectorAll()，则调用该方法查找元素，如果不支持，则模拟该方法的行为。

Sizzle 支持几乎所有的 CSS3 选择器，并且会按照文档位置返回结果。读者可以访问下面的官网，查看 Sizzle 的文档和示例：

<http://api.jquery.com/category/selectors/>

<http://sizzlejs.com/>

使用 jQuery 开发时，大多数时候，总是先调用 Sizzle 查找元素，然后调用 jQuery 方法对查找结果进行操作。此外，Sizzle 也为 jQuery 事件系统的事件代理提供基础功能（关于事件代理的相关内容请见第 9 章）。

本章主要介绍和分析 Sizzle 的设计思路、工作原理、源码实现（特别是浏览器不支持方法 querySelectorAll() 的情况），以及 jQuery 对 Sizzle 的整合和扩展。

首先来看看 Sizzle 的总体源码结构。

[⊖] [http://www.w3.org/TR/selectors-api/。](http://www.w3.org/TR/selectors-api/)

[⊖] [http://www.w3.org/TR/css3-selectors/。](http://www.w3.org/TR/css3-selectors/)

3.1 总体结构

Sizzle 的总体源码结构如代码清单 3-1 所示，为了方便解释，代码中增加了注释：

代码清单 3-1 Sizzle 的总体源码结构

```

(function() {
    // 选择器引擎入口，查找与选择器表达式 selector 匹配的元素集合
    var Sizzle = function( selector, context, results, seed ) { ... };

    // 工具方法，排序、去重
    Sizzle.uniqueSort = function( results ) { ... };

    // 便捷方法，使用指定的选择器表达式 expr 对元素集合 set 进行过滤
    Sizzle.matches = function( expr, set ) { ... };

    // 便捷方法，检查某个元素 node 是否匹配选择器表达式 expr
    Sizzle.matchesSelector = function( node, expr ) { ... };

    // 内部方法，对块表达式进行查找
    Sizzle.find = function( expr, context, isXML ) { ... };

    // 内部方法，用块表达式过滤元素集合
    Sizzle.filter = function( expr, set, inplace, not ) { ... };

    // 工具方法，抛出异常
    Sizzle.error = function( msg ) { ... };

    // 工具方法，获取 DOM 元素集合的文本内容
    var getText = Sizzle.getText = function( elem ) { ... };

    // 扩展方法和属性
    var Expr = Sizzle.selectors = {
        // 块表达式查找顺序
        order: [ "ID", "NAME", "TAG" ],
        // 正则表达式集，用于匹配和解析块表达式
        match: { ID, CLASS, NAME, ATTR, TAG, CHILD, POS, PSEUDO },
        leftMatch: { ... },
        // 属性名修正函数集
        attrMap: { "class", "for" },
        // 属性值读取函数集
        attrHandle: { href, type },
        // 块间关系过滤函数集
        relative: { "+", ">", "", "~" },
        // 块表达式查找函数集
        find: { ID, NAME, TAG },
        // 块表达式预过滤函数集
        preFilter: { CLASS, ID, TAG, CHILD, ATTR, PSEUDO, POS },
        // 伪类过滤函数集
        filters: { enabled, disabled, checked, selected, parent, empty, has, header,
                  text, radio, checkbox, file, password, submit, image, reset, button, input,
                  focus },
        // 位置伪类过滤函数集
        setFilters: { first, last, even, odd, lt, gt, nth, eq },
        // 块表达式过滤函数集
        filter: { PSEUDO, CHILD, ID, TAG, CLASS, ATTR, POS }
    };

    // 如果支持方法 querySelectorAll()，则调用该方法查找元素
    if ( document.querySelectorAll ) {
        (function() {
            var oldSizzle = Sizzle;

```

```

Sizzle = function( query, context, extra, seed ) {
    // 尝试调用方法 querySelectorAll() 查找元素
    // 如果上下文是 document，则直接调用 querySelectorAll() 查找元素
    return makeArray( context.querySelectorAll(query), extra );
    // 如果上下文是元素，则为选择器表达式增加上下文，然后调用 querySelectorAll()
    // 查找元素
    return makeArray( context.querySelectorAll( "[id='" + nid + "'] " +
        query ), extra );
    // 如果查找失败，则仍然调用 oldSizzle()
    return oldSizzle(query, context, extra, seed);
};

})();

}

// 如果支持方法 matchesSelector()，则调用该方法检查元素是否匹配选择器表达式
(function(){
    var matches = html.matchesSelector
        || html.mozMatchesSelector
        || html.webkitMatchesSelector
        || html.msMatchesSelector;
    // 如果支持方法 matchesSelector()
    if ( matches ) {
        Sizzle.matchesSelector = function( node, expr ) {
            // 尝试调用方法 matchesSelector()
            var ret = matches.call( node, expr );
            return ret;
            // 如果查找失败，则仍然调用 Sizzle()
            return Sizzle(expr, null, null, [node]).length > 0;
        };
    }
})();

// 检测浏览器是否支持 getElementsByTagName()
(function(){
    Expr.order.splice(1, 0, "CLASS");
    Expr.find.CLASS = function( match, context, isXML ) { ... };
})();
// 工具方法，检测元素 a 是否包含元素 b
Sizzle.contains = function( a, b ) { ... };
})();

```

代码清单 3-1 中的变量 Expr 与 Sizzle.selectors 指向了同一个对象，这么做是为了减少拼写字符数、缩短作用域链，并且方便压缩。但是为了直观和避免混淆，本章在描述中统一使用 Sizzle.selectors。

代码清单 3-1 中已经介绍了浏览器支持方法 querySelectorAll() 时的查找过程，本章后面的内容将介绍和分析在不支持的情况下，Sizzle 是如何模拟方法 querySelectorAll() 的行为的。另外，为了简化描述，在后文中把“块表达式查找函数集”“块表达式预过滤函数集”“块表达式过滤函数集”分别简称为“查找函数集”“预过滤函数集”“过滤函数集”。

代码清单 3-1 中的方法和属性大致可以分为 4 类：公开方法、内部方法、工具方法、扩展方法及属性。它们之间的调用关系如图 3-1 所示。

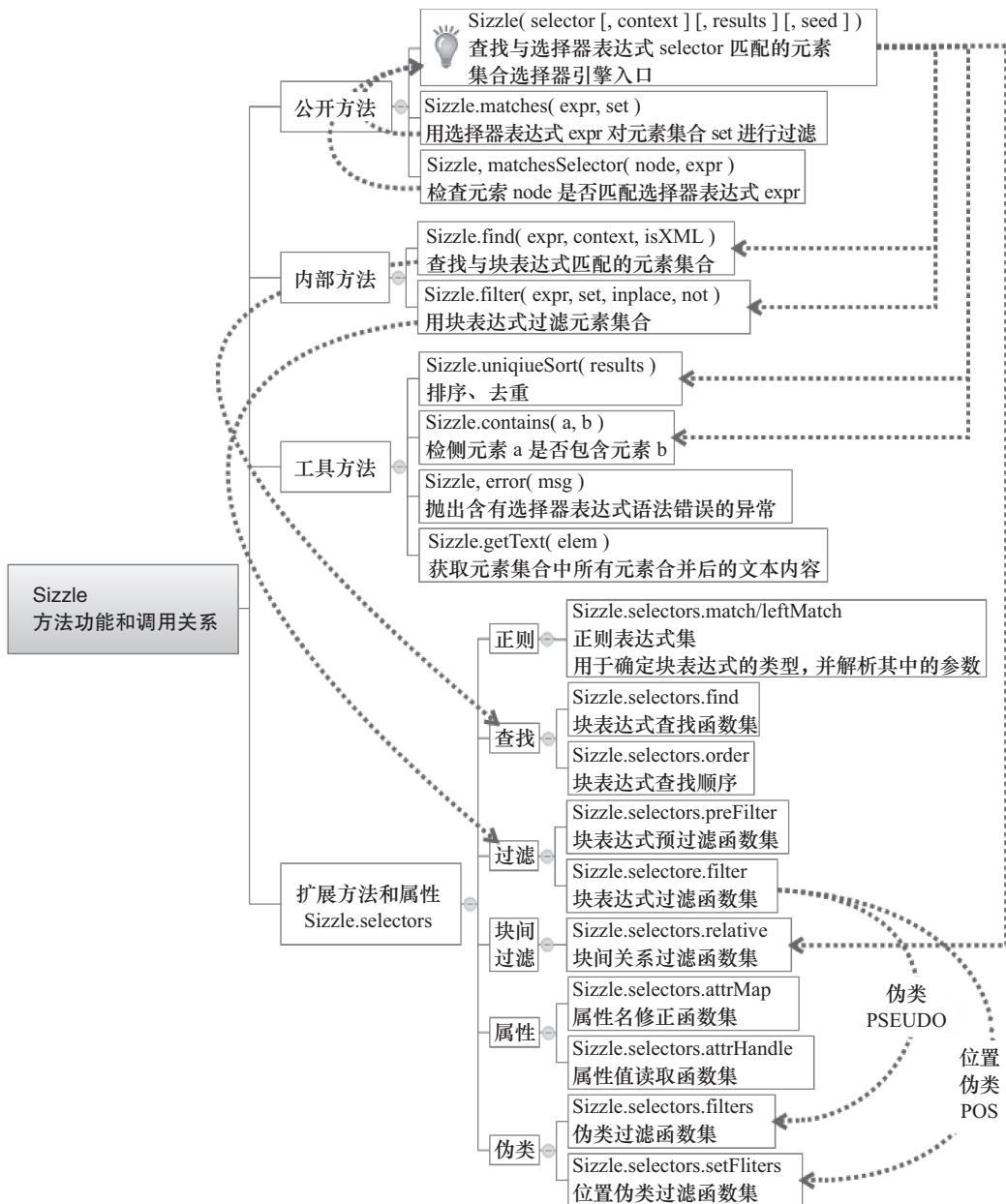


图 3-1 Sizzle 的方法、功能和调用关系

3.2 选择器表达式

为了准确描述 Sizzle 的实现，避免歧义，需要先约定一些相关术语，具体如表 3-1 所示。

表 3-1 术语和约定

序号	术 语	说明和示例
1	选择器表达式	CSS 选择器表达式, 例如, "div>p"
2	并列选择器表达式	逗号分割的多个选择器表达式, 例如, "div, p"
3	块表达式	例如, "div>p" 中的 "div"、"p"
4	块表达式类型	例如, "div" 的类型是 TAG, ".red" 的类型是 CLASS, "div.red" 则是 TAG + CLASS。共有 8 种块表达式类型: ID、CLASS、NAME、ATTR、TAG、CHILD、POS、PSEUDO
5	块间关系符	表示块表达式之间关系的符号, 例如, "div>p" 中的 ">"。共有 4 种块间关系符: ">" 父子关系、"" 祖先后代关系、"+" 紧挨着的兄弟元素、" ~ " 之后的所有兄弟元素

选择器表达式由块表达式和块间关系符组成, 如图 3-2 所示。其中, 块表达式分为 3 种: 简单表达式、属性表达式、伪类表达式; 块间关系符分为 4 种: ">" 父子关系、"" 祖先后代关系、"+" 紧挨着的兄弟元素、" ~ " 之后的所有兄弟元素; 块表达式和块间关系符组成了层级表达式。

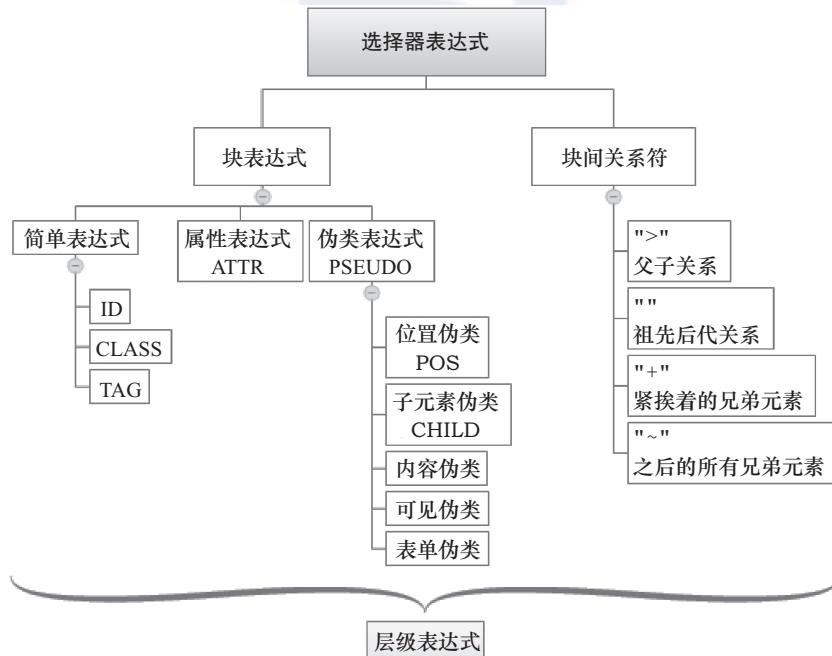


图 3-2 选择器表达式

3.3 设计思路

在正式开始分析 Sizzle 的源码实现之前, 先来讨论和分析下如果要执行一段选择器表达式, 或者说设计一个简化版的选择器引擎, 需要做些什么工作。下面以 "div.red>p" 为例来模拟执行过程, 具体来说有从左向右查找和从右向左查找两种思路:

- 1) 从左向右：先查找 "div.red" 匹配的元素集合，然后查找匹配 "p" 的子元素集合。
- 2) 从右向左：先查找 "p" 匹配的元素集合，然后检查其中每个元素的父元素是否匹配 "div.red"。

无论是从左向右还是从右向左，都必须经历下面 3 个步骤：

1) 首先要能正确地解析出 "div.red>p" 中的 "div.red"、"p" 和 ">"，即解析出选择器表达式中的块表达式和块间关系符。这一步是必需的，否则根本无从下手。

2) 然后要能正确地找到与 "div.red" 或 "p" 匹配的元素集合，即查找单个块表达式的匹配元素集合。以 "div.red" 为例，可以有两种实现方式：

- a. 先查找匹配 "div" 的元素集合，然后从中过滤出匹配 ".red" 的元素集合。
- b. 先查找匹配 ".red" 的元素集合，然后从中过滤出匹配 "div" 的元素集合。

不管采用以上哪种方式，这个过程都可以分解为两个步骤：第一步用块表达式的一部分进行查找，第二步用块表达式的剩余部分对查找的结果进行过滤。

3) 最后来处理 "div.red" 和 "p" 之间的关系符 ">"，即处理块表达式之间的父子关系。在这一步骤中，从左向右和从右向左的处理方式是截然不同的：

a. 从左向右：找到 "div.red" 匹配的元素集合的子元素集合，然后从中过滤出匹配 "p" 的子元素集合。

b. 从右向左：检查每个匹配 "p" 的元素的父元素是否匹配 "div.red"，只保留匹配的元素。

无论采用以上哪种方式，这个过程都可以分解为两个步骤：第一步按照块间关系符查找元素，第二步用块表达式对查找的结果进行过滤。不论元素之间是哪种关系（父子关系、祖先后代关系、相邻的兄弟关系或不相邻的兄弟关系），都可以采用这种方式来查找和过滤。

另外，如果还有更多的块表达式，则重复执行第 3 步。

对于前面的 3 个步骤，可以进一步提炼总结，如下：

- 1) 处理选择器表达式：解析选择器表达式中的块表达式和块间关系符。
- 2) 处理块表达式：用块表达式的一部分查找，用剩余部分对查找结果进行过滤。
- 3) 处理块间关系符：按照块间关系符查找，用块表达式对查找结果进行过滤。

从前面对选择器表达式的执行过程的分析，还可以推导分析出以下结论：

- 从左向右的总体思路是不断缩小上下文，即不断缩小查找范围。
- 从右向左的总体思路是先查找后过滤。
- 在从左向右的查找过程中，每次处理块间关系符时都需要处理未知数量的子元素或后代元素，而在从右向左的查找过程中，处理块间关系符时只需要处理单个父元素或有限数量的祖先元素。因此，在大多数情况下，采用从右向左的查找方式其效果要高于从左向右。

在了解了两种执行思路后，现在再来看看 Sizzle，它是一款从右向左查找的选择器引擎，提供了与前面 3 个步骤相对应的核心接口：

- 正则 chunker 负责从选择器表达式中提取块表达式和块间关系符。
- 方法 Sizzle.find(expr, context, isXML) 负责查找块表达式匹配的元素集合，方法 Sizzle.filter(expr, set, inplace, not) 负责用块表达式过滤元素集合。

- 对象 Sizzle.selector.relative 中的块间关系过滤函数根据块间关系符过滤元素集合。
- 函数 Sizzle(selector, context, results, seed) 则按照前面 3 个步骤将这些核心接口组织起来。本节对选择器引擎和 Sizzle 的设计思路作了探索和概述，接下来看看 Sizzle 的源码实现。

3.4 Sizzle(selector, context, results, seed)

函数 Sizzle(selector, context, results, seed) 用于查找与选择器表达式 selector 匹配的元素集合。该函数是选择器引擎的入口。

函数 Sizzle(selector, context, results, seed) 执行的 6 个关键步骤如下：

- 1) 解析块表达式和块间关系符。
- 2) 如果存在位置伪类，则从左向右查找：
 - a. 查找第一个块表达式匹配的元素集合，得到第一个上下文元素集合。
 - b. 遍历剩余的块表达式和块间关系符，不断缩小上下文元素集合。
- 3) 否则从右向左查找：
 - a. 查找最后一个块表达式匹配的元素集合，得到候选集、映射集。
 - b. 遍历剩余的块表达式和块间关系符，对映射集执行块间关系过滤。
- 4) 根据映射集筛选候选集，将最终匹配的元素放入结果集。
- 5) 如果存在并列选择器表达式，则递归调用 Sizzle(selector, context, results, seed) 查找匹配的元素集合，并合并、排序、去重。
- 6) 最后返回结果集。

下面来看看该函数的源码实现。

1. 定义 Sizzle(selector, context, results, seed)

相关代码如下所示：

```
3879 var Sizzle = function( selector, context, results, seed ) {
```

第 3879 行：定义函数 Sizzle(selector, context, results, seed)，接受 4 个参数：

- 参数 selector：CSS 选择器表达式。
- 参数 context：DOM 元素或文档对象，作为查找元素的上下文，用于限定查找范围。默认值是当前文档对象。
- 参数 results：可选的数组或类数组，函数 Sizzle(selector, context, results, seed) 将把查找到的元素添加到其中。
- 参数 seed：可选的元素集合，函数 Sizzle(selector, context, results, seed) 将从该元素集合中过滤出匹配选择器表达式的元素集合。

2. 修正参数 results、context

相关代码如下所示：

```
3880     results = results || [];
3881     context = context || document;
3882
3883     var origContext = context;
3884
```

```

3885     if ( context.nodeType !== 1 && context.nodeType !== 9 ) {
3886         return [];
3887     }
3888
3889     if ( !selector || typeof selector !== "string" ) {
3890         return results;
3891     }
3892

```

第 3880 行：如果未传入参数 results，则默认为空数组 []。方法 .find(selector) 调用 Sizzle(selector, context, results, seed) 时会传入一个 jQuery 对象，匹配元素将会被添加到传入的 jQuery 对象中。

第 3881 行：如果未传入参数 context，则默认为当前 document 对象。

第 3883 行：备份上下文 context。因为如果参数 selector 是以 #id 开头的，可能会把上下文修正为 #id 所匹配的元素。这里备份的 origContext 用于存在并列选择器表达式的情况。

第 3885 ~ 3887 行：如果参数 context 不是元素，也不是 document 对象，则忽略本次查询，直接返回空数组 []。

第 3889 ~ 3891 行：如果参数 selector 是空字符串，或者不是字符串，则忽略本次查询，直接返回传入的参数 results。

3. 定义局部变量

相关代码如下所示：

```

3893     var m, set, checkSet, extra, ret, cur, pop, i,
3894         prune = true,
3895         contextXML = Sizzle.isXML(context),
3896         parts = [],
3897         soFar = selector;
3898

```

第 3893 ~ 3897 行：定义一组局部变量，它们的含义和用途如下：

- 变量 m：用于存放正则 chunker 每次匹配选择器表达式 selector 的结果。
- 变量 set：在从右向左的查找方式中，变量 set 称为“候选集”，是最后一个块表达式匹配的元素集合，其他块表达式和块间关系符则会对候选集 set 进行过滤；对于从左向右的查找方式，变量 set 是当前块表达式匹配的元素集合，也是下一个块表达式的上下文。
- 变量 checkSet：对于从右向左的查找方式，变量 checkSet 称为“映射集”，其初始值是候选集 set 的副本，其他块表达式和块间关系符则会对映射集 checkSet 进行过滤，过滤时先根据块间关系符将其中的元素替换为父元素、祖先元素或兄弟元素，然后把与块表达式不匹配的元素替换为 false，最后根据映射集 checkSet 筛选候选集 set；对于从右向左的查找方式，事实上在查找过程中并不涉及变量 checkSet，只是在函数 Sizzle() 的最后为了统一筛选和合并匹配元素的代码，将变量 checkSet 与变量 set 指向了同一个数组。
- 变量 extra：用于存储选择器表达式中第一个逗号之后的其他并列选择器表达式。如果存

在并列选择器表达式，则会递归调用函数 Sizzle(selector, context, results, seed) 查找匹配元素集合，并执行合并、排序和去重操作。

- 变量 ret：只在从右向左执行方式中用到，用于存放查找器 Sizzle.find(expr, context, isXML) 对最后一个块表达式的查找结果，格式为 { expr: “...” , set: array }。
- 变量 pop：只在从右向左的查找方式中用到，表示单个块表达式。
- 变量 prune：只在从右向左的查找方式中用到，表示候选集 set 是否需要筛选，默认为 true，表示需要筛选，如果选择器表达式中只有一个块表达式，则变量 prune 为 false。
- 变量 contextXML：表示上下文 context 是否是 XML 文档。
- 变量 parts：存放了正则 chunker 从选择器表达式中提取的块表达式和块间关系符。
- 变量 soFar：用于保存正则 chunker 每次从选择器表达式中提取了块表达式或块间关系符后的剩余部分，初始值为完整的选择器表达式。

4. 解析块表达式和块间关系符

相关代码如下所示：

```

3860 var chunker = /((?:\(((?:\(([^\(\)]+)\|[^(\)]+)\)+\)|\[((?:\[[^\[\]]*\]|[""])*[""]|[^[\[\]]"+]\|\.\|[^>~],(\[\[\]\]+)\|[^>~])\(\s*,\s*\)?((?:.|\\r|\\n)*))/g,
3899      // Reset the position of the chunker regexp (start from head)
3900      do {
3901          chunker.exec( "" );
3902          m = chunker.exec( soFar );
3903
3904          if ( m ) {
3905              soFar = m[3];
3906
3907              parts.push( m[1] );
3908
3909              if ( m[2] ) {
3910                  extra = m[3];
3911                  break;
3912              }
3913          }
3914      } while ( m );
3915

```

第 3900 ~ 3914 行：用正则 chunker 从选择器表达式中提取块表达式和块间关系符，直到全部提取完毕或遇到下一个并列选择器表达式为止。正则 chunker 称为“分割器”，含有 3 个分组：块表达式或块间关系符、逗号、选择器表达式剩余部分，这也是 Sizzle 中最长、最复杂、最关键的正则，具体将在 3.5 节单独介绍和分析。

第 3901 ~ 3902 行：正则 chunker 每次匹配选择器表达式的剩余部分之前，先通过匹配一个空字符串来重置正则 chunker 的开始匹配位置，从而使得每次匹配时都会从头开始匹配。直接设置“chunker.lastIndex = 0;”也能达到同样的效果。

第 3904 ~ 3913 行：如果正则 chunker 可以匹配选择器表达式的剩余部分，则将第三个分组（即经过当前匹配后的剩余部分）赋予变量 soFar，下次 do-while 循环时继续匹配；通过这种方式也可过滤掉一些垃圾字符（如空格）；同时，将第一个分组中的块表达式或块间关系

符插入数组 parts 中；此外，如果第二个分组不是空字符串，即遇到了逗号，表示接下来是一个并列选择器表达式，则将第三个分组保存在变量 extra，然后结束循环。

5. 如果存在位置伪类，则从左向右查找

相关代码如下所示：

```

3916     if ( parts.length > 1 && origPOS.exec( selector ) ) {
3917
3918     if ( parts.length === 2 && Expr.relative[ parts[0] ] ) {
3919         set = posProcess( parts[0] + parts[1], context, seed );
3920
3921     } else {
3922         set = Expr.relative[ parts[0] ] ?
3923             [ context ] :
3924             Sizzle( parts.shift(), context );
3925
3926     while ( parts.length ) {
3927         selector = parts.shift();
3928
3929         if ( Expr.relative[ selector ] ) {
3930             selector += parts.shift();
3931         }
3932
3933         set = posProcess( selector, set, seed );
3934     }
3935 }
3936
4221 var Expr = Sizzle.selectors = {
4222     match: [
4231         POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\((\d*)\))?(?=([^\-]|$))/,
4233     ],
4749 },
4751 var origPOS = Expr.match.POS,
```

第 3916 ~ 3935 行：如果存在块间关系符（即相邻的块表达式之间有依赖关系）和位置伪类，例如，\$('div button:first')，则从左向右查找。正则 origPOS 中定义了所支持的位置伪类，见第 4231 行。

为什么遇到位置伪类需要从左向右查找呢？以 \$("div button:first") 为例，在查找所有 div 元素下的所有 button 元素中的第一个时，位置伪类 ":first" 过滤的是 "div button" 匹配的元素集合，因此，必须从左向右查找，并且需要先从选择器表达式中删除位置伪类，然后执行查找，最后才用位置伪类过滤查找结果。这个过程由函数 posProcess(selector, context, seed) 实现。

第 3918 ~ 3919 行：如果数组 parts 中只有两个元素，并且第一个是块间关系符，则可以直接调用函数 posProcess(selector, context, seed) 查找匹配的元素集合。

第 3921 ~ 3935 行：否则，从左向右对数组 parts 中的其他块表达式逐个进行查找，每次查找时指定前一个块表达式匹配的元素集合作为上下文，即不断缩小查找范围。

第 3922 ~ 3924 行：首先，从数组 parts 头部弹出第一个块表达式，递归调用函数 Sizzle(selector, context, results, seed) 查找匹配元素集合，得到第一个上下文元素集合；如果数组 parts 的第一个元素是块间关系符，则直接把参数 context 作为第一个上下文元素集合。

第 3926 ~ 3934 行：从左向右遍历数组 parts 中的其他块表达式和块间关系符，调用函数 posProcess(selector, context, seed) 查找匹配元素集合，调用时传入的参数 selector 含有一个块间关系符和一个块表达式，并且指定上下文为前一个块表达式匹配的元素集合 set，调用后再将返回值赋值给变量 set，作为下一个块表达式的上下文。

posProcess(selector, context, seed)

函数 posProcess(selector, context, seed) 在指定的上下文数组 context 下，查找与选择器表达式 selector 匹配的元素集合，并且支持位置伪类。选择器表达式 selector 由一个块间关系符和一个块表达式组成。

函数 posProcess(selector, context, seed) 执行的 3 个关键步骤如下：

- 1) 删除选择器表达式中的所有伪类。
- 2) 调用 Sizzle(selector, context, results, seed) 查找删除伪类后的选择器表达式所匹配的元素集合。
- 3) 调用 Sizzle.filter(expr, set, inplace, not) 用伪类过滤查找结果。

下面来看看该函数的源码实现。相关代码如下所示：

```

5266 var posProcess = function( selector, context, seed ) {
5267     var match,
5268         tmpSet = [],
5269         later = "",
5270         root = context.nodeType ? [context] : context,
5271
5272     //Position selectors must be done after the filter
5273     //And so must :not(positional) so we move all PSEUDOS to the end
5274     while ( (match = Expr.match.PSEUDO.exec( selector )) ) {
5275         later += match[0];
5276         selector = selector.replace( Expr.match.PSEUDO, "" );
5277     }
5278
5279     selector = Expr.relative[selector] ? selector + "*" : selector;
5280
5281     for ( var i = 0, l = root.length; i < l; i++ ) {
5282         Sizzle( selector, root[i], tmpSet, seed );
5283     }
5284
5285     return Sizzle.filter( later, tmpSet );
5286 };

```

第 5274 ~ 5277 行：删除选择器表达式中的所有伪类，并累计在变量 later 中。

第 5279 行：如果删除伪类后的选择器表达式只剩一个块间关系符，则追加一个通配符 “**”。

第 5281 ~ 5283 行：遍历上下文数组，调用函数 Sizzle(selector, context, results, seed) 查找删除伪类后的选择器表达式匹配的元素集合，将查找结果合并到数组 tmpSet 中。

第 5285 行：调用方法 Sizzle.filter(expr, set, inplace, not) 用记录的伪类 later 过滤元素集合 tmpSet，并返回一个新数组，其中只包含了过滤后的元素。

下面回到对函数 Sizzle(selector, context, results, seed) 的分析中。

6. 如果不存在位置伪类，则从右向左查找

(1) 尝试缩小查找范围

相关代码如下所示：

```

3937     } else {
3938         // Take a shortcut and set the context if the root selector is an ID
3939         // (but not if it'll be faster if the inner selector is an ID)
3940         if ( !seed && parts.length > 1 &&
3941             context.nodeType === 9 &&
3942             !contextXML &&
3943             Expr.match.ID.test(parts[0]) &&
3944             !Expr.match.ID.test(parts[parts.length - 1]) ) {
3945             ret = Sizzle.find( parts.shift(), context, contextXML );
3946             context = ret.expr ?
3947                 Sizzle.filter( ret.expr, ret.set )[0] :
3948                 ret.set[0];
3949             if ( context ) {
3950                 // 省略从右向左查找的代码
3951             } else {
3952                 checkSet = parts = [];
3953             }
3954         }
3955     }

```

第 3940 ~ 3947 行：如果第一个块选择器是 ID 类型（即格式为 #id），并且最后一个块选择器不是 ID 类型，则修正上下文 context 为第一个块选择器匹配的元素，以缩小查找范围，提高查找效率。在这个过程中，先调用方法 Sizzle.find(expr, context, isXML) 对第一个块表达式执行简单的查找，如果还有剩余部分，再调用方法 Sizzle.filter(expr, set, inplace, not) 对查找结果进行过滤，最后取匹配元素集合的第一个元素作为后续查找的上下文。

第 3982 ~ 3984 行：如果第一个块表达式是 ID 类型，但是没有找到匹配的元素，则没有继续查找和过滤的必要了，此时直接清空数组 parts，并设置映射集为空数组。

(2) 查找最后一个块表达式匹配的元素集合，得到候选集 set、映射集 checkSet

相关代码如下所示：

```

3949         if ( context ) {
3950             ret = seed ?
3951                 { expr: parts.pop(), set: makeArray(seed) } :
3952                 Sizzle.find( parts.pop(), parts.length === 1 && (parts[0] === "~" ||
parts[0] === "+") && context.parentNode ? context.parentNode : context, contextXML );
3953             set = ret.expr ?
3954                 Sizzle.filter( ret.expr, ret.set ) :
3955                 ret.set;
3956             if ( parts.length > 0 ) {
3957                 checkSet = makeArray( set );
3958             }
3959         }
3960     }

```

```

3961         } else {
3962             prune = false;
3963         }
3964

```

第 3950 ~ 3956 行：查找最后一个块表达式匹配的元素集合，得到候选集 set。先调用方法 Sizzle.find(expr, context, isXML) 对最后一个块表达式执行简单的查找，如果还有剩余部分，再调用方法 Sizzle.filter(expr, set, inplace, not) 对查找结果进行过滤。

如果传入了参数 seed，则不需要调用 Sizzle.find() 查找，只调用 Sizzle.filter() 过滤。

第 3958 ~ 3963 行：如果数组 parts 中还有其他元素，即还有块表达式或块间关系符，则创建一份候选集 set 的副本，并赋值给 checkSet，作为映射集；如果数组 parts 为空，则表示选择器表达式中只有一个块表达式，此时设置变量 prune 为 false，表示不需要对候选集 set 进行筛选。

(3) 遍历剩余的块表达式和块间关系符，对映射集 checkSet 执行块间关系过滤

相关代码如下所示：

```

3965     while ( parts.length ) {
3966         cur = parts.pop();
3967         pop = cur;
3968
3969         if ( !Expr.relative[ cur ] ) {
3970             cur = "";
3971         } else {
3972             pop = parts.pop();
3973         }
3974
3975         if ( pop == null ) {
3976             pop = context;
3977         }
3978
3979         Expr.relative[ cur ]( checkSet, pop, contextXML );
3980     }
3981

```

第 3965 ~ 3980 行：从右向左遍历数组 parts 中剩余的块表达式和块间关系符，调用块间关系符在 Sizzle.selectors.relative 中对应的过滤函数，对映射集 checkSet 执行块间关系过滤，直至数组 parts 为空为止。

第 3966 ~ 3973 行：变量 cur 表示块间关系符，变量 pop 表示块间关系符左侧的块表达式。每次遍历时，如果弹出的元素不是块间关系符，则默认为后代关系符；如果弹出的元素是块间关系符，则再弹出一个作为块表达式。因为是从右向左查找，所以变量 pop 的作用是作为过滤映射集 checkSet 的上下文。

第 3975 ~ 3977 行：如果仍然未找到前一个块表达式 pop，则表示已经到达数组头部，直接将上下文 context 作为映射集 checkSet 的上下文。

第 3979 行：块间关系过滤函数的参数格式为：

```
Sizzle.selectors.relative[ 块间关系符 cur ]( 映射集 checkSet, 左侧块表达式 pop, contextXML );
```

在块间关系过滤函数中，会先根据块间关系符 cur 的类型将映射集 checkSet 的元素替换为父元素、祖先元素或兄弟元素，然后将与左侧块表达式 pop 不匹配的元素替换为 false，具体请参见 3.8 节。

7. 根据映射集 checkSet 筛选候选集 set，将最终的匹配元素放入结果集 results

相关代码如下所示：

```

3987     if ( !checkSet ) {
3988         checkSet = set;
3989     }
3990
3991     if ( !checkSet ) {
3992         Sizzle.error( cur || selector );
3993     }
3994
3995     if ( toString.call(checkSet) === "[object Array]" ) {
3996         if ( !prune ) {
3997             results.push.apply( results, checkSet );
3998
3999         } else if ( context && context.nodeType === 1 ) {
4000             for ( i = 0; checkSet[i] != null; i++ ) {
4001                 if ( checkSet[i] && (checkSet[i] === true || checkSet[i].
nodeType === 1 && Sizzle.contains(context, checkSet[i])) ) {
4002                     results.push( set[i] );
4003                 }
4004             }
4005
4006         } else {
4007             for ( i = 0; checkSet[i] != null; i++ ) {
4008                 if ( checkSet[i] && checkSet[i].nodeType === 1 ) {
4009                     results.push( set[i] );
4010                 }
4011             }
4012         }
4013
4014     } else {
4015         makeArray( checkSet, results );
4016     }
4017

```

第 3987 ~ 3989 行：在前面查找匹配元素集合的过程中，如果是从左向右查找的，不会涉及映射集 checkSet；如果是从右向左查找的，且只有一个块表达式，也不会对于映射集 checkSet 赋值。在这两种情况下，为了统一筛选和合并匹配元素的代码，在这里要先设置映射集 checkSet 和候选集 set 指向同一个数组。

第 3995 ~ 4012 行：如果映射集 checkSet 是数组，则遍历映射集 checkSet，检查其中的元素是否满足匹配条件，如果满足，则将候选集 set 中对应的元素放入结果集 results。

第 3996 ~ 3997 行：如果变量 prune 为 false，表示不需要筛选候选集 set，则直接将映射集 checkSet 插入结果集 results 中。注意这里的隐藏逻辑：当选择器表达式中只有一个块表达式时，才会设置变量 prune 为 false，此时映射集 checkSet 和候选集 set 指向同一个数组，见第 3958 ~ 3963 行、第 3987 ~ 3989 行的说明。

第 3999 ~ 4004 行：如果上下文是元素，而不是文档对象，则遍历映射集 checkSet，如果其中的元素满足以下条件之一，则将候选集 set 中对应的元素放入结果集 results：

- 是 true。
- 是元素，并且包含在上下文 context 中。

第 4006 ~ 4012 行：如果上下文是文档对象，则遍历映射集 checkSet，如果其中的元素满足以下全部条件，则将候选集 set 中对应的元素放入结果集 results：

- 不是 null。
- 是元素。

第 4014 ~ 4016 行：如果候选集 checkSet 不是数组，则可能是 NodeList，这种情况只会在选择器表达式仅仅是简单的标签或类样式（如 \$("div")、\$(".red")）时才会出现，此时不需要筛选候选集 set，并且映射集 checkSet 和候选集 set 会指向同一个数组，可以直接将映射集 checkSet 插入结果集 results 中。见第 3958 ~ 3963 行、第 3987 ~ 3989 行的说明。

8. 如果存在并列选择器表达式，则递归调用 Sizzle(selector, context, results, seed) 查找匹配的元素集合，并合并、排序、去重

相关代码如下所示：

```
4018     if ( extra ) {
4019         Sizzle( extra, origContext, results, seed );
4020         Sizzle.uniqueSort( results );
4021     }
4022 }
```

第 4020 行：方法 Sizzle.uniqueSort(results) 负责对元素集合中的元素排序、去重，具体请参见 3.10.1 节。

9. 最后返回结果集 results

相关代码如下所示：

```
4023     return results;
4024 }
```

函数 Sizzle(selector, context, results, seed) 的执行过程可以总结为图 3-3。

3.5 正则 chunker

正则 chunker 用于从选择器表达式中提取块表达式和块间关系符。该正则是 Sizzle 中最长、最复杂和最关键的正则，图 3-4 是该正则的分解图，图中包含了每个子块的功能介绍和测试用例。

3.6 Sizzle.find(expr, context, isXML)

方法 Sizzle.find(expr, context, isXML) 负责查找与块表达式匹配的元素集合。该方法会按照表达式类型数组 Sizzle.selectors.order 规定的查找顺序（ID、CLASS、NAME、TAG）逐个尝试查找，如果未找到，则查找上下文的所有后代元素（*）。

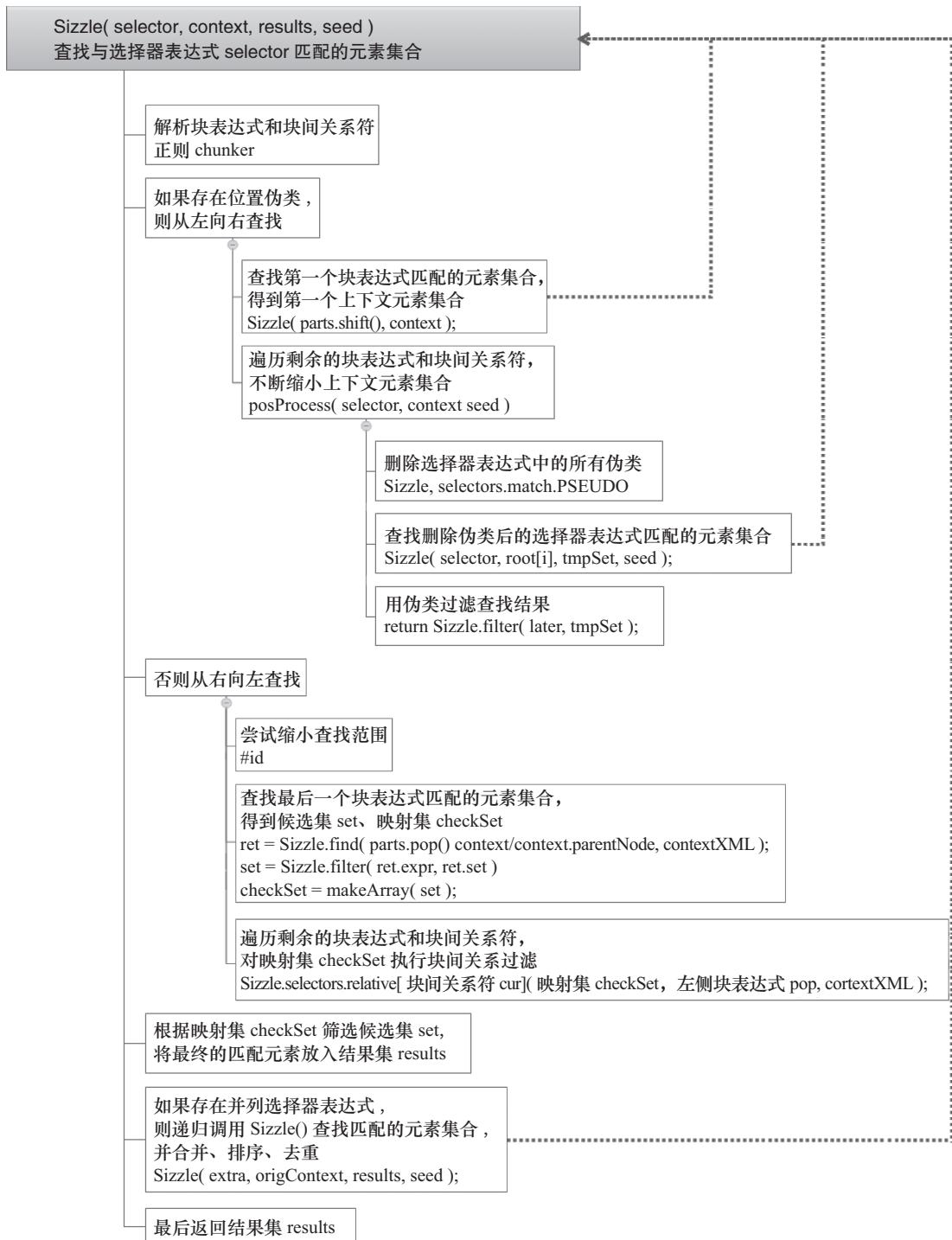


图 3-3 Sizzle(selector, context, results, seed) 的执行过程

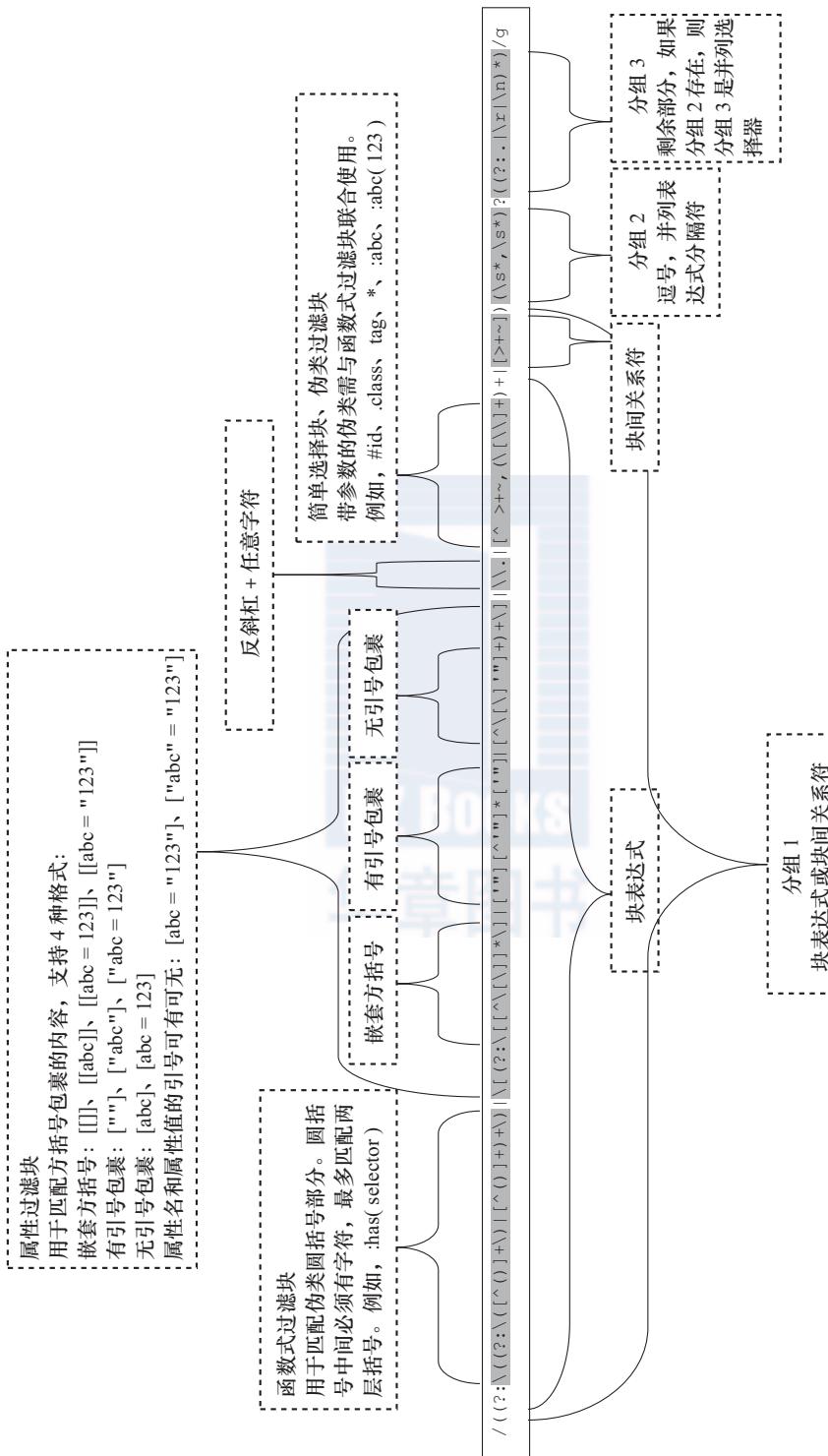


图 3-4 正则 chunker

方法 Sizzle.find(expr, context, isXML) 执行的 5 个关键步骤如下：

- 1) 先用正则集合 Sizzle.selectors.leftMatch 中的正则确定表达式类型。
- 2) 然后调用查找函数集 Sizzle.selectors.find 中对应类型的查找函数，查找匹配的元素集合。
- 3) 然后删除块表达式中已查找过的部分。
- 4) 如果没有找到对应类型的查找函数，则读取上下文的所有后代元素。
- 5) 最后返回格式为 {set: 候选集 , expr: 块表达式剩余部分} 的对象。

下面来看看该方法的源码实现。

1. 定义 Sizzle.find(expr, context, isXML)

相关代码如下所示：

```
4051 Sizzle.find = function( expr, context, isXML ) {
```

第 4051 行：定义方法 Sizzle.find(expr, context, isXML)，它接受 3 个参数：

- 参数 expr：块表达式。
- 参数 context：DOM 元素或文档对象，作为查找时的上下文。
- 参数 isXML：布尔值，指示是否运行在一个 XML 文档中。

2. 遍历表达式类型数组 Sizzle.selectors.order

相关代码如下所示：

```
4052     var set, i, len, match, type, left;
4053
4054     if ( !expr ) {
4055         return [];
4056     }
4057
4058     for ( i = 0, len = Expr.order.length; i < len; i++ ) {
4059         type = Expr.order[i];
4060     }
```

第 4054 ~ 4056 行：如果块表达式 expr 是空字符串，则直接返回空数组 []。

第 4058 行：表达式类型数组 Sizzle.selectors.order 中定义了查找单个块表达式时的顺序，依次是 ID、CLASS、NAME、TAG，其中，CLASS 需要浏览器支持方法 getElementsByClassName()。关于 Sizzle.selectors.order 的具体说明请参见 3.9.1 节。

(1) 确定块表达式类型 Sizzle.selectors.leftMatch[type]

相关代码如下所示：

```
4061         if ( (match = Expr.leftMatch[ type ].exec( expr )) ) {
4062             left = match[1];
4063             match.splice( 1, 1 );
4064         }
```

第 4061 行：检查每个表达式类型 type 在 Sizzle.selectors.leftMatch 中对应的正则是否匹配块表达式 expr，如果匹配，则可以确定块表达式的类型。

第 4062 ~ 4063 行：对象 Sizzle.selectors.leftMatch 中存放了表达式类型和正则的映射，正则可以用于确定块表达式的类型，并解析其中的参数。它是基于对象 Sizzle.selectors.match

初始化的，具体请参见 3.9.2 节。

(2) 查找匹配元素 Sizzle.selectors.find[type]

相关代码如下所示：

```
4065     if ( left.substr( left.length - 1 ) !== "\\" ) {
4066         match[1] = (match[1] || "").replace( rBackslash, "" );
4067         set = Expr.find[ type ]( match, context, isXML );
4068     }
```

第 4065 行：如果匹配正则的内容以反斜杠 "\\" 开头，表示反斜杠 "\\" 之后的字符被转义了，不是期望的类型，这时会认为类型匹配失败。

第 4066 行：过滤其中的反斜杠，以支持将某些特殊字符（例如，“#”、“.”、“[”）作为普通字符使用。举个例子，假设某个 input 元素的属性 id 是 "a.b"，则对应的选择器表达式应该写作 \$("#a\\l.b")，这里替换掉了反斜杠，又会变回 "a.b"，因此仍然可以通过执行 document.getElementById("a.b") 查找到 input 元素。

第 4067 行：调用表达式类型 type 在查找函数集 Sizzle.selectors.find 中对应的查找函数，查找匹配的元素集合。Sizzle.selectors.find 中定义了 ID、CLASS、NAME、TAG 所对应的查找函数，具体请参见 3.9.3 节。

(3) 删除块表达式中已查找过的部分

相关代码如下所示：

```
4069     if ( set != null ) {
4070         expr = expr.replace( Expr.match[ type ], "" );
4071         break;
4072     }
4073 }
4074 }
4075 }
4076 }
```

第 4069 ~ 4072 行：如果 set 不是 null 和 undefined，表示对应的查找函数执行了查找操作，则不管有没有找到匹配元素，都将块表达式 expr 中已查找过的部分删除，并结束方法 Sizzle.find(expr, context, isXML) 的查找过程。

第 4070 行：用对象 Sizzle.selectors.match 中对应的正则来匹配已查找过的部分，具体请参见 3.9.2 节。

3. 如果没有找到对应类型的查找函数，则读取上下文的所有后代元素

相关代码如下所示：

```
4077     if ( !set ) {
4078         set = typeof context.getElementsByTagName != "undefined" ?
4079             context.getElementsByTagName( "*" ) :
4080             [];
4081     }
4082 }
```

第 4077 ~ 4081 行：如果 set 是 null 或 undefined，（大多数情况下）表示没有在 Sizzle.

selectors.find 中找到对应的查找函数，例如，`$(".input")` 会读取上下文的所有后代元素作为候选集。

4. 返回 { set: 候选集 , expr: 块表达式剩余部分 }

相关代码如下所示：

```
4083     return { set: set, expr: expr };
4084 };
```

方法 Sizzle.find(expr, context, isXML) 的执行过程可总结为图 3-5。

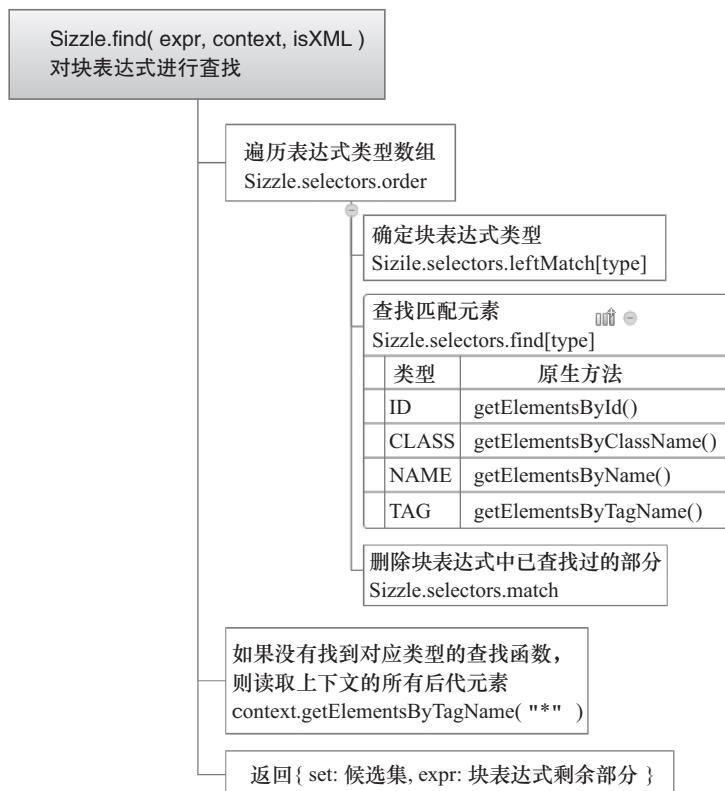


图 3-5 Sizzle.find(expr, context, isXML) 的执行过程

3.7 Sizzle.filter(expr, set, inplace, not)

方法 Sizzle.filter(expr, set, inplace, not) 负责用块表达式过滤元素集合。在该方法内部，将用过滤函数集 Sizzle.selectors.filter 中的过滤函数来执行过滤操作。

方法 Sizzle.filter(expr, set, inplace, not) 实现的 5 个关键步骤如下：

- 1) 首先用正则集合 Sizzle.selectors.leftMatch 中的正则确定块表达式类型。
- 2) 然后调用预过滤函数集 Sizzle.selectors.preFilter 中对应类型的预过滤函数，执行过滤前的修正操作。

3) 调用过滤函数集 Sizzle.selectors.filter[type] 中对应类型的过滤函数，执行过滤操作，如果过滤函数返回 false，则把元素集合中对应位置的元素替换为 false。

4) 最后删除块表达式中已过滤的部分。

5) 重复第 1) ~ 4) 步，直至块表达式变为空字符串。

下面来看看该方法的源码实现。

1. 定义 Sizzle.filter(expr, set, inplace, not)

相关代码如下所示：

```
4086 Sizzle.filter = function( expr, set, inplace, not ) {
```

第 4086 行：定义方法 Sizzle.filter(expr, set, inplace, not)，它接受 4 个参数：

- 参数 expr：块表达式。
- 参数 set：待过滤的元素集合。
- 参数 inplace：布尔值。如果为 true，则将元素集合 set 中与选择器表达式不匹配的元素设置为 false；如果不为 true，则重新构造一个元素数组并返回，只保留匹配元素。
- 参数 not：布尔值。如果为 true，则去除匹配元素，保留不匹配元素；如果不为 true，则去除不匹配元素，保留匹配元素。

2. 用块表达式 expr 过滤元素集合 set，直到 expr 为空

相关代码如下所示：

```
4087     var match, anyFound,
4088         type, found, item, filter, left,
4089         i, pass,
4090         old = expr,
4091         result = [],
4092         curLoop = set,
4093         isXMLFilter = set && set[0] && Sizzle.isXML( set[0] );
4094
4095     while ( expr && set.length ) {
```

第 4095 行：用块表达式 expr 过滤元素集合 set，直到 expr 变为空字符串。如果候选集 set 变为空数组，则没有必要继续执行过滤操作。

(1) 遍历块过滤函数集 Sizzle.selectors.filter

相关代码如下所示：

```
4096         for ( type in Expr.filter ) {
```

第 4096 行：遍历块过滤函数集 Sizzle.selectors.filter，调用其中的过滤函数执行过滤操作。块过滤函数集 Sizzle.selectors.filter 中定义了 PSEUDO、CHILD、ID、TAG、CLASS、ATTR、POS 对应的过滤函数，具体请参见 3.9.7 节。

(2) 确定块表达式类型 Sizzle.selectors.leftMatch[type]

相关代码如下所示：

```
4097             if ( (match = Expr.leftMatch[ type ].exec( expr )) != null && match[2] ) {
```

```

4098         filter = Expr.filter[ type ];
4099         left = match[1];
4100
4101         anyFound = false;
4102
4103         match.splice(1,1);
4104
4105         if ( left.substr( left.length - 1 ) === "\\" ) {
4106             continue;
4107         }
4108

```

第 4097 行：检查每个表达式类型 type 在 Sizzle.selectors.leftMatch 中对应的正则是否匹配块表达式 expr，如果匹配，则可以确定块表达式的类型。

第 4105 ~ 4107 行：如果匹配正则的内容以反斜杠 "\\" 开头，表示反斜杠 "\\" 之后的字符被转义了，不是期望的类型，这时会认为类型匹配失败。

(3) 调用预过滤函数 Sizzle.selectors.preFilter[type]

相关代码如下所示：

```

4109         if ( curLoop === result ) {
4110             result = [];
4111         }
4112
4113         if ( Expr.preFilter[ type ] ) {
4114             match = Expr.preFilter[ type ]( match, curLoop, inplace,
result, not, isXMLFilter );
4115
4116             if ( !match ) {
4117                 anyFound = found = true;
4118
4119             } else if ( match === true ) {
4120                 continue;
4121             }
4122         }
4123

```

第 4109 ~ 4111 行：用于缩小候选集。

第 4113 ~ 4122 行：如果在预过滤函数集 Sizzle.selectors.preFilter 中存在对应的预过滤函数，则调用，执行过滤前的修正操作。预过滤函数负责进一步修正过滤参数，具体请参见 3.9.4 节。

第 4116 ~ 4121 行：预过滤函数有 3 种返回值：

- false：已经执行了过滤，缩小了候选集，例如，CLASS。
- true：需要继续执行预过滤，尚不到执行过滤函数的时候，例如，POS、CHILD。
- 字符串：修正后的过滤参数（通常是块表达式），后面会继续调用对应的过滤函数。

(4) 调用过滤函数 Sizzle.selectors.filter[type]

相关代码如下所示：

```
4124         if ( match ) {
```

```

4125         for ( i = 0; (item = curLoop[i]) != null; i++ ) {
4126             if ( item ) {
4127                 found = filter( item, match, i, curLoop );
4128                 pass = not ^ found;
4129
4130                 if ( inplace && found != null ) {
4131                     if ( pass ) {
4132                         anyFound = true;
4133
4134                     } else {
4135                         curLoop[i] = false;
4136                     }
4137
4138                 } else if ( pass ) {
4139                     result.push( item );
4140                     anyFound = true;
4141                 }
4142             }
4143         }
4144     }
4145

```

第 4124 ~ 4144 行：遍历元素集合 curLoop，对其中的每个元素执行过滤函数，检测元素是否匹配。

第 4127 ~ 4128 行：变量 found 表示当前元素是否匹配过滤表达式；变量 pass 表示当前元素 item 是否可以通过过滤表达式的过滤。如果变量 found 为 true，表示匹配，此时如果未指定参数 not，则变量 pass 为 true；如果变量 found 为 false，表示不匹配，此时如果参数 not 为 true，则变量 pass 为 true；其他情况下，变量 pass 为 false。

第 4130 ~ 4141 行：如果参数 inplace 为 true，则将与块表达式 expr 不匹配的元素设置为 false；如果参数 inplace 不是 true，则重新构造一个元素数组，只保留匹配元素，即会不断缩小元素集合。

(5) 删除块表达式 expr 中已过滤的部分

相关代码如下所示：

```

4146         if ( found !== undefined ) {
4147             if ( !inplace ) {
4148                 curLoop = result;
4149             }
4150
4151             expr = expr.replace( Expr.match[ type ], "" );
4152
4153             if ( !anyFound ) {
4154                 return [];
4155             }
4156
4157             break;
4158         }
4159     }
4160 }
4161

```

第 4146 行：变量 `found` 是过滤函数 `Sizzle.selectors.filter[type]` 的返回值，如果不等于 `undefined`，表示至少执行过一次过滤。大多数情况下，过滤操作发生在过滤函数中，不过也可能发生在预过滤函数中，例如，`CLASS`、`POS`、`CHILD`。

第 4147 ~ 4149 行：如果参数 `inplace` 不是 `true`，则将新构建的元素数组赋值给变量 `curLoop`，在下次循环时，会将 `result` 再次置为空数组（见第 4109 ~ 4111 行），然后存放通过过滤的元素（见第 4130 ~ 4141 行），然后再赋值给变量 `curLoop`，即会不断地缩小元素集合。

第 4151 行：删除块表达式中已过滤过的部分，直至块表达式变为空字符串。用对象 `Sizzle.selectors.match` 中对应的正则匹配已过滤过的部分，具体请参见 3.9.2 节。

第 4153 ~ 4155 行：如果没有找到可以通过过滤的元素，直接返回一个空数组。

(6) 如果块表达式没有发生变化，则认为不合法

相关代码如下所示：

```

4162     // Improper expression
4163     if ( expr === old ) {
4164         if ( anyFound == null ) {
4165             Sizzle.error( expr );
4166         } else {
4167             break;
4168         }
4169     }
4170 }
4171
4172     old = expr;
4173 }
4174

4178 Sizzle.error = function( msg ) {
4179     throw new Error( "Syntax error, unrecognized expression: " + msg );
4180 };

```

第 4163 ~ 4172 行：如果块表达式 `expr` 没有发生变化，说明前面的过滤没有生效，动不了块表达式 `expr` 分毫，此时如果没有找到可以通过过滤的元素，则认为块表达式 `expr` 不合法，抛出语法错误的异常。

3. 返回过滤后的元素集合，或缩减范围后的元素集合

相关代码如下所示：

```

4175     return curLoop;
4176 };

```

方法 `Sizzle.filter(expr, set, inplace, not)` 的执行过程可以总结为图 3-6。

3.8 Sizzle.selectors.relative

对象 `Sizzle.selectors.relative` 中存放了块间关系符和对应的块间关系过滤函数，称为“块间关系过滤函数集”。

块间关系符共有 4 种，其含义和过滤方式如表 3-2 所示。



图 3-6 Sizzle.filter(expr, set, inplace, not) 的执行过程

表 3-2 块间关系符的含义和过滤方式

序号	块间关系符	选择器表达式	说 明	从右向左的过滤方式
1	""	ancestor descendant	匹配所有后代元素	检查祖先元素是否匹配左侧的块表达式
2	"+"	prev + next	匹配下一个兄弟元素	检查前一个兄弟元素否匹配左侧的块表达式
3	parent > child	匹配所有子元素	检查父元素是否匹配左侧的块表达式	
4	" ~ "	prev ~ siblings	匹配之后的所有兄弟元素	检查之前的兄弟元素是否匹配左侧的块表达式

在函数 Sizzle(selector, context, results, seed) 从右向左进行过滤时，块间关系过滤函数被调用，用于检查映射集 checkSet 中的元素是否匹配块间关系符左侧的块表达式。调用时的参数格式为：

```
Sizzle.selectors.relative[ 块间关系符 cur ]( 映射集 checkSet, 左侧块表达式 pop,
contextXML );
```

块间关系过滤函数接受 3 个参数：

- 参数 checkSet：映射集，对该元素集合执行过滤操作。
- 参数 part：大多数情况下是块间关系符左侧的块表达式，该参数也可以是 DOM 元素。
- 参数 isXML：布尔值，指示是否运行在一个 XML 文档中。

块间关系过滤函数实现的 3 个关键步骤如下：

- 1) 遍历映射集 checkSet。
- 2) 按照块间关系符查找每个元素的兄弟元素、父元素或祖先元素。
- 3) 检查找到的元素是否匹配参数 part，并替换映射集 checkSet 中对应位置的元素。
 - a. 如果参数 part 是标签，则检查找到的元素其节点名称 nodeName 是否与之相等，如果相等则替换为找到的元素，不相等则替换为 false。
 - b. 如果参数 part 是 DOM 元素，则检查找到的元素是否与之相等，如果相等则替换为 true，不相等则替换为 false。
 - c. 如果参数 part 是非标签字符串，则调用方法 Sizzle.filter(selector, set, inplace, not) 过滤。也就是说，遍历结束后，映射集 checkSet 中的元素可能会是兄弟元素、父元素、祖先元素、true 或 false。

3.8.1 "+"

块间关系符 "+" 匹配选择器 "prev + next"，即匹配所有紧接在元素 prev 后的兄弟元素 next。例如，\$("div + span")、\$(".lastdiv + span")。对于从右向左的查找方式，则是检查元素 next 之前的兄弟元素是否匹配块表达式 prev。

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4222
4223     relative: {
4224         "+": function(checkSet, part) {
4225             var isPartStr = typeof part === "string",
4226                 isTag = isPartStr && !rNonWord.test( part ),
4227                 isPartStrNotTag = isPartStr && !isTag;
4228
4229             if ( isTag ) {
4230                 part = part.toLowerCase();
4231             }
4232
4233             for ( var i = 0, l = checkSet.length, elem; i < l; i++ ) {
4234                 if ( (elem = checkSet[i]) ) {
4235                     while ( (elem = elem.previousSibling) && elem.nodeType !== 1 ) {
4236                         checkSet[i] = isPartStrNotTag || elem && elem.nodeType
4237                         .toLowerCase() === part ?
4238                             elem || false :
4239                             elem === part;
4240
4241             }
4242
4243         }
4244     }
4245 }
```

```

4268         }
4269     }
4270
4271     if ( isPartStrNotTag ) {
4272         Sizzle.filter( part, checkSet, true );
4273     }
4274 },
4338 },
4749 };

```

第 4253 ~ 4255 行：定义一组局部变量，它们的含义和用途如下：

- 变量 isPartStr：指示参数 part 是否是字符串。
- 变量 isTag：指示参数 part 是否为标签字符串。
- 变量 isPartStrNotTag：指示参数 part 是否是非标签字符串。

第 4261 ~ 4269 行：遍历映射集 checkSet，查找每个元素的前一个兄弟元素，并替换映射集 checkSet 中对应位置的元素，有以下 3 个逻辑分支：

- 如果未找到兄弟元素，则替换为 false。
- 如果找到了兄弟元素，并且参数 part 是标签，则检查兄弟元素的节点名称 nodeName 是否与之相等，如果相等则替换为兄弟元素，不相等则替换为 false。
- 如果找到了兄弟元素，并且参数 part 是 DOM 元素，则检查二者是否相等，如果相等则替换为 true，不相等则替换为 false。

因此，在遍历结束后，映射集 checkSet 中的元素可能会是兄弟元素、true 或 false。

第 4263 行：在遍历兄弟元素的同时过滤掉非元素节点，并且只要取到一个兄弟元素就退出 while 循环。

第 4271 ~ 4273 行：如果参数 part 是非标签字符串，则调用方法 Sizzle.filter(selector, set, inplace, not) 过滤映射集 checkSet。对于参数 part 是标签和 DOM 元素的情况，在前面遍历映射集 checkSet 时已经处理过了。

3.8.2 ">"

块间关系符 ">" 用于选择器 "parent > child"，即匹配父元素 parent 下的子元素 child。例如，\$("div + span")、\$(".lastdiv + span")。对于从右向左的查找方式，则是检查子元素 child 的父元素是否匹配块表达式 parent。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4222     ">": function( checkSet, part ) {
4223         var elem,
4224             isPartStr = typeof part === "string",
4225             i = 0,
4226             l = checkSet.length;

```

```

4281
4282     if ( isPartStr && !rNonWord.test( part ) ) {
4283         part = part.toLowerCase();
4284
4285         for ( ; i < l; i++ ) {
4286             elem = checkSet[i];
4287
4288             if ( elem ) {
4289                 var parent = elem.parentNode;
4290                 checkSet[i] = parent.nodeName.toLowerCase() === part ?
4291                     parent : false;
4292             }
4293         }
4294     } else {
4295         for ( ; i < l; i++ ) {
4296             elem = checkSet[i];
4297
4298             if ( elem ) {
4299                 checkSet[i] = isPartStr ?
4300                     elem.parentNode :
4301                     elem.parentNode === part;
4302             }
4303         }
4304
4305         if ( isPartStr ) {
4306             Sizzle.filter( part, checkSet, true );
4307         }
4308     }
4309 },
4338 },
4749 };

```

第 4282 ~ 4292 行：如果参数 part 是标签，则遍历映射集 checkSet，查找每个元素的父元素，并检查父元素的节点名称 nodeName 是否与参数 part 相等，如果相等则替换映射集 checkSet 中对应位置的元素为父元素，不相等则替换为 false。

第 4294 ~ 4307 行：如果参数 part 不是标签，则可能是非标签字符串或 DOM 元素，同样遍历映射集 checkSet，查找每个元素的父元素，并替换映射集 checkSet 中对应位置的元素，在这个过程中有以下 2 个逻辑分支：

- 如果参数 part 是非标签字符串，则在遍历映射集 checkSet 的过程中，替换映射集 checkSet 中对应位置的元素为父元素，遍历结束后调用方法 Sizzle.filter(selector, set, inplace, not) 过滤映射集 checkSet。
- 如果参数 part 是元素，则在遍历映射集 checkSet 时，检查每个元素的父元素是否与之相等，如果相等则替换映射集 checkSet 中对应位置的元素为 true，不相等则替换为 false。

因此，在遍历结束后，映射集 checkSet 中的元素可能会是父元素、true 或 false。

3.8.3 " "

块间关系符 " " 用于选择器 "ancestor descendant"，即匹配祖先元素 ancestor 的所有后代元素 descendant。例如，`$("#div button")`、`($("#div .btn")`)。对于从右向左的查找方式，则是检查后代元素 descendant 的祖先元素是否匹配块表达式 ancestor。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4222
4223     "": function(checkSet, part, isXML) {
4224         var nodeCheck,
4225             doneName = done++,
4226             checkFn = dirCheck;
4227
4228         if ( typeof part === "string" && !rNonWord.test( part ) ) {
4229             part = part.toLowerCase();
4230             nodeCheck = part;
4231             checkFn = dirNodeCheck;
4232         }
4233         checkFn( "parentNode", part, doneName, checkSet, nodeCheck, isXML );
4234     },
4235
4236     "[",
4237
4238     ],
4239
4240     ":",
4241
4242     "=",
4243
4244     "!=",
4245
4246     "<",
4247
4248     ">",
4249 };

```

第 4312 ~ 4322 行：这段代码含有 2 个逻辑分支：

- 如果参数 part 是非标签字符串或 DOM 元素，则调用函数 dirCheck() 过滤映射集 checkSet。
- 如果参数 part 是标签，则调用函数 dirNodeCheck() 过滤映射集 checkSet。

调用函数 dirCheck() 和 dirNodeCheck() 时的参数格式为：

```
checkFn( 方向 "parentNode/previousSibling", 块表达式 part, 缓存计数器 doneName, 映射集 checkSet, nodeCheck, isXML )
```

函数 dirCheck() 和 dirNodeCheck() 会遍历映射集 checkSet，查找每个元素的祖先元素，并检查是否有祖先元素匹配参数 part，同时替换映射集 checkSet 中对应位置的元素。具体请参见 3.8.5 节和 3.8.6 节。

3.8.4 " ~ "

块间关系符 " ~ " 用于选择器 "prev ~ siblings"，即匹配元素 prev 之后的所有兄弟元素 siblings。例如，`($('div ~ p')`)。对于从右向左的查找方式，则是检查元素 siblings 之前的兄弟元素是否匹配块表达式 prev。

`Sizzle.selectors.relative[" ~ "](checkSet, part)` 的源码实现与 `Sizzle.selectors.relative[""]` (`checkSet, part`) 几乎一样，两者的区别仅仅在于调用函数 dirCheck() 和 dirNodeCheck() 时第

一个参数的值不同，前者是 "previousSibling"，后者则是 "parentNode"。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4225     "~": function( checkSet, part, isXML ) {
4226         var nodeCheck,
4227             doneName = done++,
4228             checkFn = dirCheck;
4229
4330         if ( typeof part === "string" && !rNonWord.test( part ) ) {
4331             part = part.toLowerCase();
4332             nodeCheck = part;
4333             checkFn = dirNodeCheck;
4334         }
4335
4336         checkFn( "previousSibling", part, doneName, checkSet, nodeCheck,
4337             isXML );
4338     },

```

3.8.5 dirCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)

函数 dirCheck(dir, cur, doneName, checkSet, nodeCheck, isXML) 负责遍历候选集 checkSet，检查其中每个元素在某个方向 dir 上是否有与参数 cur 匹配或相等的元素。如果找到，则将候选集 checkSet 中对应位置的元素替换为找到的元素或 true；如果未找到，则替换为 false。

在块间关系过滤函数 Sizzle.selectors.relative["~/~"]([checkSet, part]) 中，当参数 part 是非标签字符串或 DOM 元素时，才会调用函数 dirCheck()。

相关代码如下所示：

```

5201 function dirCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
5202     for ( var i = 0, l = checkSet.length; i < l; i++ ) {
5203         var elem = checkSet[i];
5204
5205         if ( elem ) {
5206             var match = false;
5207
5208             elem = elem[dir];
5209
5210             while ( elem ) {
5211                 if ( elem[ expando ] === doneName ) {
5212                     match = checkSet[elem.sizset];
5213                     break;
5214                 }
5215
5216                 if ( elem.nodeType === 1 ) {
5217                     if ( !isXML ) {
5218                         elem[ expando ] = doneName;
5219                         elem.sizset = i;
5220                     }
5221

```

```

5222         if ( typeof cur !== "string" ) {
5223             if ( elem === cur ) {
5224                 match = true;
5225                 break;
5226             }
5227         } else if ( Sizzle.filter( cur, [elem] ).length > 0 ) {
5228             match = elem;
5229             break;
5230         }
5231     }
5232 }
5233 elem = elem[dir];
5234 }
5235 checkSet[i] = match;
5236 }
5237 }
5238 }
5239 }
5240 }

```

第 5201 行：定义函数 dirCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)，它接受 6 个参数：

- 参数 dir：表示查找方向的字符串，例如，“parentNode”、“previousSibling”。
- 参数 cur：大多数情况下是非标签字符串格式的块表达式，也可能是 DOM 元素。
- 参数 doneName：数值。本次查找的唯一标识，用于优化查找过程，避免重复查找。
- 参数 checkSet：候选集，在查找过程中，其中的元素将被替换为父元素、祖先元素、兄弟元素、true 或 false。
- 参数 nodeCheck：undefined。在后面的代码中没有用到该参数。
- 参数 isXML：布尔值，指示是否运行在一个 XML 文档中。

第 5202 ~ 5239 行：遍历候选集 checkSet，对其中的每个元素，沿着某个方向（例如，“parentNode”、“previousSibling”）一直查找，直到找到与参数 cur (DOM 元素) 相等或者与参数 cur (非标签字符串) 匹配的元素为止，或者直到在该方向上不再有元素为止。

如果找到与参数 cur (DOM 元素) 相等的元素，则替换映射集 checkSet 中对应位置的元素为 true；如果找到与参数 cur (非标签字符串) 匹配的元素，则替换为找到的元素；如果未找到，则默认替换为 false。

第 5211 ~ 5220 行：在查找过程中，如果遇到已经检查过的元素，则直接取该元素在候选集 checkSet 中对应位置上的元素，避免重复查找。

第 5222 ~ 5231 行：如果参数 cur 是 DOM 元素，则直接检查找到的元素是否与之相等；如果参数 cur 是非标签字符串，则调用方法 Sizzle.filter(expr, set, inplace, not) 检查是否与之匹配。

第 5237 行：替换映射集 checkSet 中对应位置的元素。变量 match 的初始值为 false；如果找到与参数 cur (DOM 元素) 相等的元素，则其值变为 true；如果找到与参数 cur (非标签字符串) 匹配的元素，则其值变为找到的元素。

3.8.6 dirNodeCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)

函数 dirNodeCheck(dir, cur, doneName, checkSet, nodeCheck, isXML) 负责遍历候选集 checkSet，检查其中每个元素在某个方向 dir 上是否有与参数 cur 匹配的元素。如果找到，则将候选集 checkSet 中对应位置的元素替换为找到的元素；如果未找到，则替换为 false。

在块间关系过滤函数 Sizzle.selectors.relative[""/"~"]([checkSet, part]) 中，当参数 part 是标签时，才会调用函数 dirNodeCheck()。

相关代码如下所示：

```

5168 function dirNodeCheck( dir, cur, doneName, checkSet, nodeCheck, isXML ) {
5169     for ( var i = 0, l = checkSet.length; i < l; i++ ) {
5170         var elem = checkSet[i];
5171
5172         if ( elem ) {
5173             var match = false;
5174
5175             elem = elem[dir];
5176
5177             while ( elem ) {
5178                 if ( elem[ expando ] === doneName ) {
5179                     match = checkSet[elem.sizset];
5180                     break;
5181                 }
5182
5183                 if ( elem.nodeType === 1 && !isXML ) {
5184                     elem[ expando ] = doneName;
5185                     elem.sizset = i;
5186                 }
5187
5188                 if ( elem.nodeName.toLowerCase() === cur ) {
5189                     match = elem;
5190                     break;
5191                 }
5192
5193                 elem = elem[dir];
5194             }
5195
5196             checkSet[i] = match;
5197         }
5198     }
5199 }
```

第 5168 行：定义函数 dirNodeCheck(dir, cur, doneName, checkSet, nodeCheck, isXML)，它接受 6 个参数：

- 参数 dir：表示查找方向的字符串，例如，“parentNode”、“previousSibling”。
- 参数 cur：标签字符串。
- 参数 doneName：数值。本次查找的唯一标识，用于优化查找过程，避免重复查找。
- 参数 checkSet：候选集，在查找过程中，其中的元素将被替换为与参数 cur 匹配的元素或 false。

□ 参数 nodeCheck：标签字符串。在后边的代码中没有用到该参数。

□ 参数 isXML：布尔值，指示是否运行在一个 XML 文档中。

第 5169 ~ 5198 行：遍历候选集 checkSet，对其中的每个元素，沿着某个方向（例如，“parentNode”、“previousSibling”）一直查找，直到找到节点名称 nodeName 与参数 cur 相等的元素，或者在该方向上不再有元素为止。如果找到，则替换映射集 checkSet 中对应位置的元素为找到的元素；如果未找到，则默认替换为 false。

第 5178 ~ 5186 行：在查找过程中，如果遇到已经检查过的元素，则直接取该元素在候选集 checkSet 中对应位置上的元素，避免重复查找。

第 5196 行：替换映射集 checkSet 中对应位置的元素。变量 match 初始值为 false，如果找到节点名称 nodeName 与参数 cur 相等的元素，则其值变为找到的元素。

3.9 Sizzle.selectors

对象 Sizzle.selectors 包含了 Sizzle 在查找和过滤过程中用到的正则、查找函数、过滤函数，其中包含的属性见图 3-1，源码结构见代码清单 3-1。

3.9.1 Sizzle.selectors.order

表达式类型数组 Sizzle.selectors.order 中定义了查找单个块表达式时的查找顺序，依次是 ID、CLASS、NAME、TAG。其中，CLASS 需要浏览器支持方法 getElementsByTagName。查找顺序综合考虑了浏览器是否支持、查找结果集的大小、查找效率、使用频率等因素。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4222     order: [ "ID", "NAME", "TAG" ],
4223
4224     // Opera can't find a second classname (in 9.6)
4225     // Also, make sure that getElementsByTagName actually exists
4226     if ( !div.getElementsByTagName || div.getElementsByTagName("e") .
length === 0 ) {
4227         return;
4228     }
4229
4230     // Safari caches class attributes, doesn't catch changes (in 3.2)
4231     div.lastChild.className = "e";
4232
4233     if ( div.getElementsByTagName("e").length === 1 ) {
4234         return;

```

```

5155      }
5156
5157      Expr.order.splice(1, 0, "CLASS");
5158      Expr.find.CLASS = function( match, context, isXML ) {
5159          if ( typeof context.getElementsByClassName !== "undefined" && !isXML ) {
5160              return context.getElementsByClassName(match[1]);
5161          }
5162      };
5163
5164      // release memory in IE
5165      div = null;
5166  })();

```

第 5140 ~ 5155 行：测试当前浏览器是否正确支持方法 `getElementsByClassName()`。测试思路是先构造一段 DOM 结构，然后调用方法 `getElementsByClassName()`，检查是否返回期望数量的元素。如果不支持或不能正确支持，则不做任何事情。

第 5157 ~ 5162 行：如果当前浏览器支持方法 `getElementsByClassName()`，则：

- 向 `Sizzle.selectors.order` 中插入 "CLASS"，由 ["ID", "NAME", "TAG"] 变为 ["ID", "CLASS", "NAME", "TAG"]，插入位置在 "ID" 之后、"NAME" 之前。
- 向 `Sizzle.selectors.find` 中插入 "CLASS" 对应的查找函数。

3.9.2 Sizzle.selectors.match/leftMatch

对象 `Sizzle.selectors.match/leftMatch` 中存放了表达式类型和正则的映射，正则用于确定块表达式的类型，并解析其中的参数。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4222
4223     match: {
4224         ID: /#((?:[\w\u00c0-\uFFFF\-\-]|\.\.)+)/,
4225         CLASS: /\.(?:[\w\u00c0-\uFFFF\-\-]|\.\.)+/,
4226         NAME: /\[name=[\'\"]*(?:[\w\u00c0-\uFFFF\-\-]|\.\.)+)[\'\"]\*/,
4227         ATTR: /\[\s*(?:[\w\u00c0-\uFFFF\-\-]|\.\.)+\)\s*(?:(\S?=)\s*(?:([\'"])
4228         (*.*?\)\3|(#?(?:[\w\u00c0-\uFFFF\-\-]|\.\.)*)|))\s*\]\*/,
4229         TAG: /^(?:[\w\u00c0-\uFFFF\*\-\-]|\.\.)+$/,
4230         CHILD: /:(only|nth|last|first)-child(?:\(\s*(even|odd|(?:[+\-]?\d+
4231         (?:[+\-]?\d*)?n\s*(?:[+\-]\s*\d+)?))\s*\))/,
4232         POS: /:(nth|eq|gt|lt|first|last|even|odd)(?:\(((\d*)\))?(?:=[^\-]|$)/,
4233         PSEUDO: /:(?:[\w\u00c0-\uFFFF\-\-]|\.\.)+)(?:\(([\'\"]?)((?:\([^\)]+\)|[^\\()])+
4234         *\))+\2\))/,
4235     },
4236
4237     4749 },
4240
4241     origPOS = Expr.match.POS,
4242     fescape = function(all, num) {
4243         return "\\" + (num - 0 + 1);
4244     };
4245
4246     4756 for ( var type in Expr.match ) {

```

```

4757     Expr.match[ type ] = new RegExp( Expr.match[ type ].source + /(?![^\\[]*\\])  

        (?![^\\(.)*\\]))/.source ) ;  

4758     Expr.leftMatch[ type ] = new RegExp( /(^(?:.|\\r|\\n)*?)/.source + Expr.  

match[ type ].source.replace(/\\((\\d+)/g, fescape) ) ;  

4759 }

```

第 4224 ~ 4233 行：定义一组正则，稍后会逐个分析和测试。

第 4756 ~ 4759 行：为对象 Sizzle.selectors.match 中的正则增加一段后缀正则 /(?![^\\[]*\\])
(?![^\\(.)*\\]))/，然后再加上一段前缀正则 /(^(?:.|\\r|\\n)*?)/，来构造对象 Sizzle.selectors.leftMatch
中的同名正则。因为增加的前缀正则中包含了一个分组，所以原正则中的分组编号需要加 1
后移。

1. 后缀正则 /(?![^\\[]*\\])(?![^\\(.)*\\])/

后缀正则 /(?![^\\[]*\\])(?![^\\(.)*\\])/ 要求接下来的字符不能含有 "]"、")"，用于确保选择器
表达式的语法正确，以及确保正确匹配嵌套选择器表达式。

例如，执行 \$("input[name=foo\\.baz]") 时会抛出语法异常，因为选择器表达式 "input
[name=foo\\.baz]" 的末尾多了一个 "]"，对象 Sizzle.selectors.match/leftMatch 中没有正则可以
匹配其中的 "[name=foo\\.baz]"；如果没有后缀正则，则 Sizzle.selectors.match/leftMatch.
NAME 会匹配 "[name=foo\\.baz]"，并执行查找，而不会抛出语法异常。

又如，执行 \$("input[name=foo.baz]") 时会抛出语法异常，因为选择器表达式 "input
[name=foo.baz]" 没有转义点号，对象 Sizzle.selectors.match/leftMatch 中没有正则可以匹配其中的 "[name=foo.baz]"；如果没有后缀正则，则对象 Sizzle.selectors.match/leftMatch.CLASS 会
匹配 "input[name=foo.baz]" 中的 ".baz"，并执行查找，然后用 "input[name=foo]" 过滤查找结果，
而不会抛出语法异常。

再如，执行 \$("input:not(.foo)") 时，会先查找匹配 "input" 的元素集合，然后从中过滤
不匹配 ".foo" 的元素集合；如果没有后缀正则，则会变为先查找匹配 ".foo" 的元素集合，
然后从中过滤匹配 "input:not()" 的元素集合。

读者可以将第 4757 行代码注释掉，然后测试和验证上述例子。

2. 前缀正则 /(^(?:.|\\r|\\n)*?)/

前缀正则 /(^(?:.|\\r|\\n)*?)/ 用于捕获匹配正则的表达式之前的字符，主要是捕获转义反斜
杠，以支持将特殊字符作为普通字符使用。

例如，".test\\#id"，在用正则 Sizzle.selectors.match.ID 匹配时会发现 "#" 之前是转义反
斜杠 "\\"，这时将认为该表达式的类型不是 ID；如果没有前缀正则，则会先查找匹配 "#id"
的元素集合，然后从中过滤出匹配 ".test\\" 的元素集合。

接下来分析和测试对象 Sizzle.selectors.match 中 ID、CLASS、NAME、ATTR、TAG、
CHILD、POS、PSEUDO 对应的正则。测试用例参考自 Sizzle 的测试用例 <https://github.com/jquery/sizzle/blob/1.7.1/test/unit/selector.js>。

3. ID

正则 Sizzle.selector.match.ID 用于匹配简单表达式 "#id"，并解析 "#" 之后的字符串，

其中含有 1 个分组：id。解析图见图 3-7，测试用例见表 3-3。

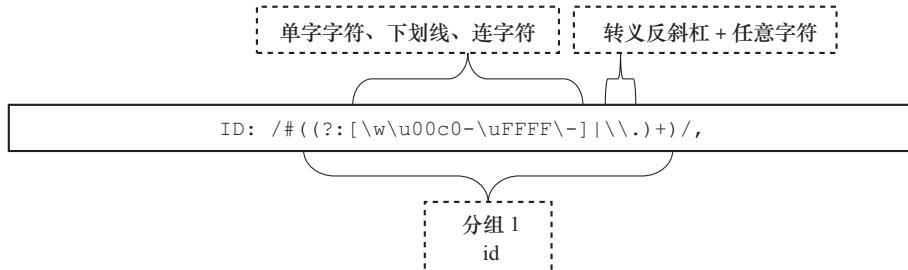


图 3-7 正则 Sizzle.selectors.match.ID

表 3-3 正则 Sizzle.selectors.match.ID

序号	测试用例	运行结果
1	ID.exec("#id")	["#id", "id"]
2	ID.exec("#firstp#simon1")	["#firstp", "firstp"]
3	ID.exec("#台北 Táiběi")	["# 台北 Táiběi", "台北 Táiběi"]
4	ID.exec("#foo\\:bar")	["#foo\\:bar", "foo\\:bar"]
5	ID.exec("#test\\.foo\\[5\\]bar")	["#test\\.foo\\[5\\]bar", "test\\.foo\\[5\\]bar"]

4. CLASS

正则 Sizzle.selector.match.CLASS 用于匹配简单表达式 ".class"，并解析 "." 之后的字符串，其中含有 1 个分组：类样式。解析图见图 3-8，测试用例见表 3-4。

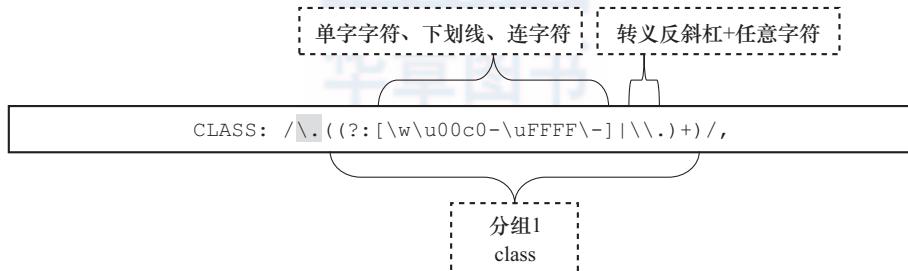


图 3-8 正则 Sizzle.selectors.match.CLASS

表 3-4 正则 Sizzle.selectors.match.CLASS

序号	测试用例	运行结果
1	CLASS.exec(".blog")	[".blog", "blog"]
2	CLASS.exec(".blog.link")	[".blog", "blog"]
3	CLASS.exec(".台北 Táiběi")	[". 台北 Táiběi", "台北 Táiběi"]
4	CLASS.exec(".foo\\:bar")	[".foo\\:bar", "foo\\:bar"]
5	CLASS.exec(".test\\.foo\\[5\\]bar")	[".test\\.foo\\[5\\]bar", "test\\.foo\\[5\\]bar"]

5. NAME

正则 Sizzle.selector.match.NAME 用于匹配属性表达式 "[name = "value"]"，并解析属性 name 的值，其中含有 1 个分组：属性 name 的值。解析图见图 3-9，测试用例见表 3-5。

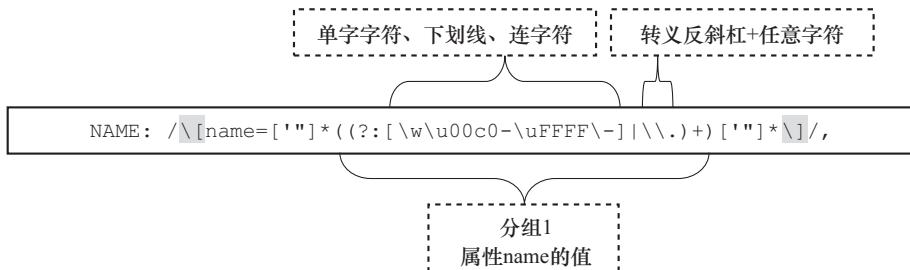


图 3-9 正则 Sizzle.selector.match.NAME

表 3-5 正则 Sizzle.selector.match.NAME

序号	测试用例	运行结果
1	NAME.exec("input[name=action]")	[["name=action"], "action"]
2	NAME.exec("input[name=' action ']")	[["name=' action '"], "action"]
3	NAME.exec("input[name=\"action\"]")	[["name=\"action\""], "action"]
4	NAME.exec("input[name=\"types[]\"]")	null

6. ATTR

正则 Sizzle.selector.match.ATTR 用于匹配属性表达式 "[attribute = "value"]"，并解析属性名和属性值，其中含有 5 个分组：属性名、等号部分、引号、属性值、无引号时的属性值。解析图见图 3-10，测试用例见表 3-6。

7. TAG

正则 Sizzle.selector.match.TAG 用于匹配简单表达式 "tag"，并解析标签名，其中含有 1 个分组：标签名。解析图见图 3-11，测试用例见表 3-7。

8. CHILD

正则 Sizzle.selector.match.CHILD 用于匹配子元素伪类表达式 :nth-child(index/even/odd/equation)、:first-child、:last-child、:only-child，并解析子元素伪类和伪类参数，其中含有 2 个分组：子元素伪类、伪类参数。解析图见图 3-12，测试用例见表 3-8。

9. POS

正则 Sizzle.selector.match.POS 用于匹配位置伪类表达式 ":eq(index)"、":gt(index)"、":lt(index)"、":first"、":last"、":odd"、":even"，并解析位置伪类和伪类参数，其中含有 2 个分组：位置伪类、伪类参数。解析图见图 3-13，测试用例见表 3-9。

10. PSEUDO

正则 Sizzle.selector.match.PSEUDO 用于匹配伪类表达式，请解析 ":" 之后的伪类和伪类参数，其中含有 3 个分组：伪类、引号、伪类参数。解析图见图 3-14，测试用例见表 3-10。

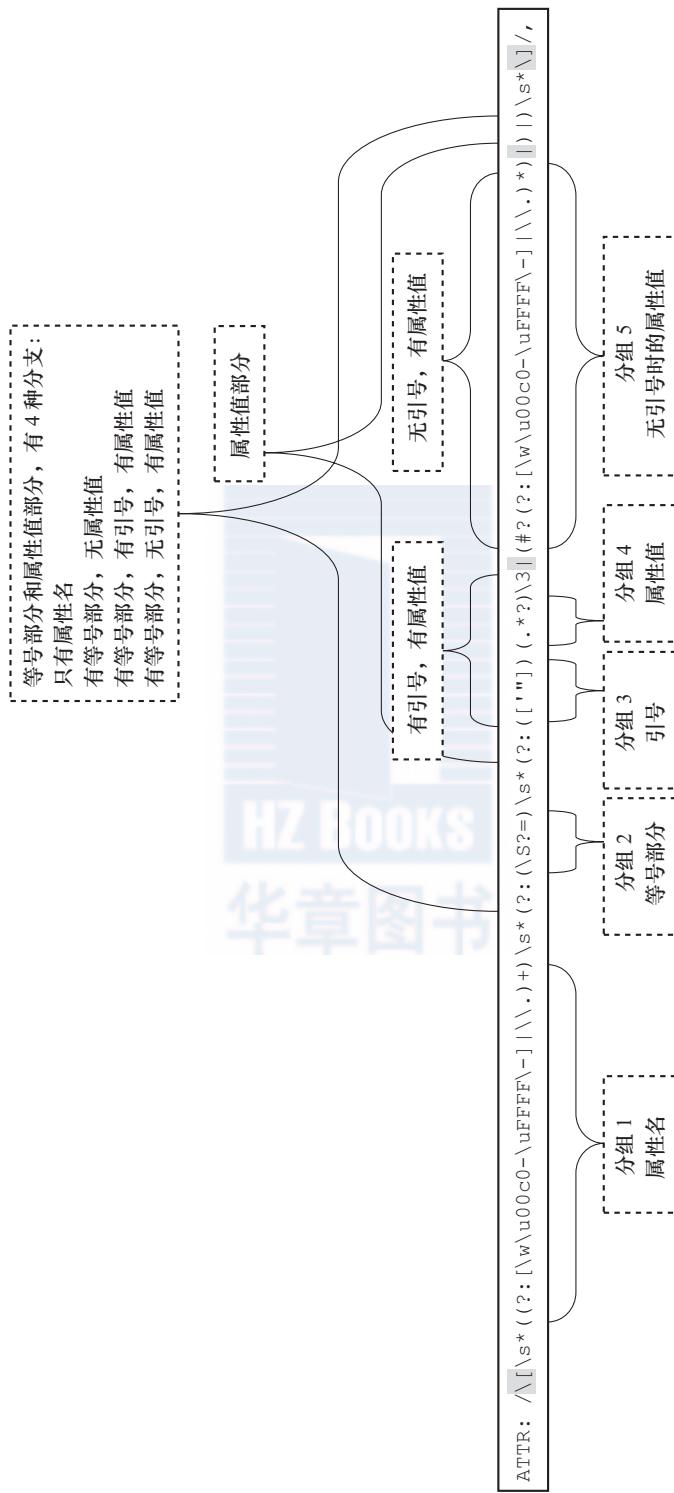


图 3-10 正则 Sizzle.selectors.match.ATTR

表 3-6 正则 Sizzle.selectors.match.ATTR

序号	测试用例	运行结果
1	ATTR.exec("a[title]")	["[title]", "title", undefined, undefined, undefined]
2	ATTR.exec("a[title=]")	["[title]", "title", "=", undefined, undefined, ""]
3	ATTR.exec("a[rel='bookmark ']")	["[rel='bookmark ']", "rel", "=", " ", "bookmark", undefined]
4	ATTR.exec("a[rel=\"bookmark\"]")	["[rel=\"bookmark\"]", "rel", "=", " ", "bookmark", undefined]
5	ATTR.exec("a[rel=bookmark]")	["[rel=bookmark]", "rel", "=", undefined, undefined, "bookmark"]
6	ATTR.exec("a[rel='bookmark']")	["[rel='bookmark']", "rel", "=", " ", "bookmark", undefined]
7	ATTR.exec("input[name=foo\\.baz]")	["[name=foo\\.baz]", "name", "=", undefined, undefined, "foo\\.baz"]
8	ATTR.exec("input[name=foo\\\[baz\\]]")	["[name=foo\\\[baz\\]]", "name", "=", undefined, undefined, "foo\\\[baz\\]"]
9	ATTR.exec("a[href='http://www.google.com/]")	["[href='http://www.google.com/']", "href", "=", " ", "http://www.google.com/", undefined]
10	ATTR.exec("a[href^='http://www ']")	["[href^='http://www ']", "href", "^=", " ", "http://www", undefined]
11	ATTR.exec("a[href\$='org/]")	["[href\$='org']", "href", "\$=", " ", "org", undefined]
12	ATTR.exec("a[href*= 'google ']")	["[href*= 'google ']", "href", "*=", " ", "google", undefined]
13	ATTR.exec("option[value=' ']")	["[value=' ']", "value", "=", " ", " ", undefined]
14	ATTR.exec("option[value!= ' ']")	["[value!= ' ']", "value", "!=" , " ", " ", undefined]
15	ATTR.exec("[xml\\:test]")	["[xml\\:test]", "xml\\:test", undefined, undefined, undefined, undefined]
16	ATTR.exec("[data-foo]")	["[data-foo]", "data-foo", undefined, undefined, undefined, undefined]

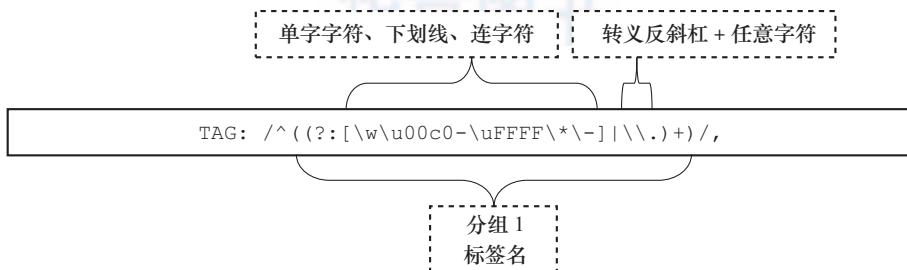


图 3-11 正则 Sizzle.selectors.match.TAG

表 3-7 正则 Sizzle.selectors.match.TAG

序号	测试用例	运行结果
1	TAG.exec("body")	["body", "body"]
2	TAG.exec("html")	["html", "html"]
3	TAG.exec("h1")	["h1", "h1"]

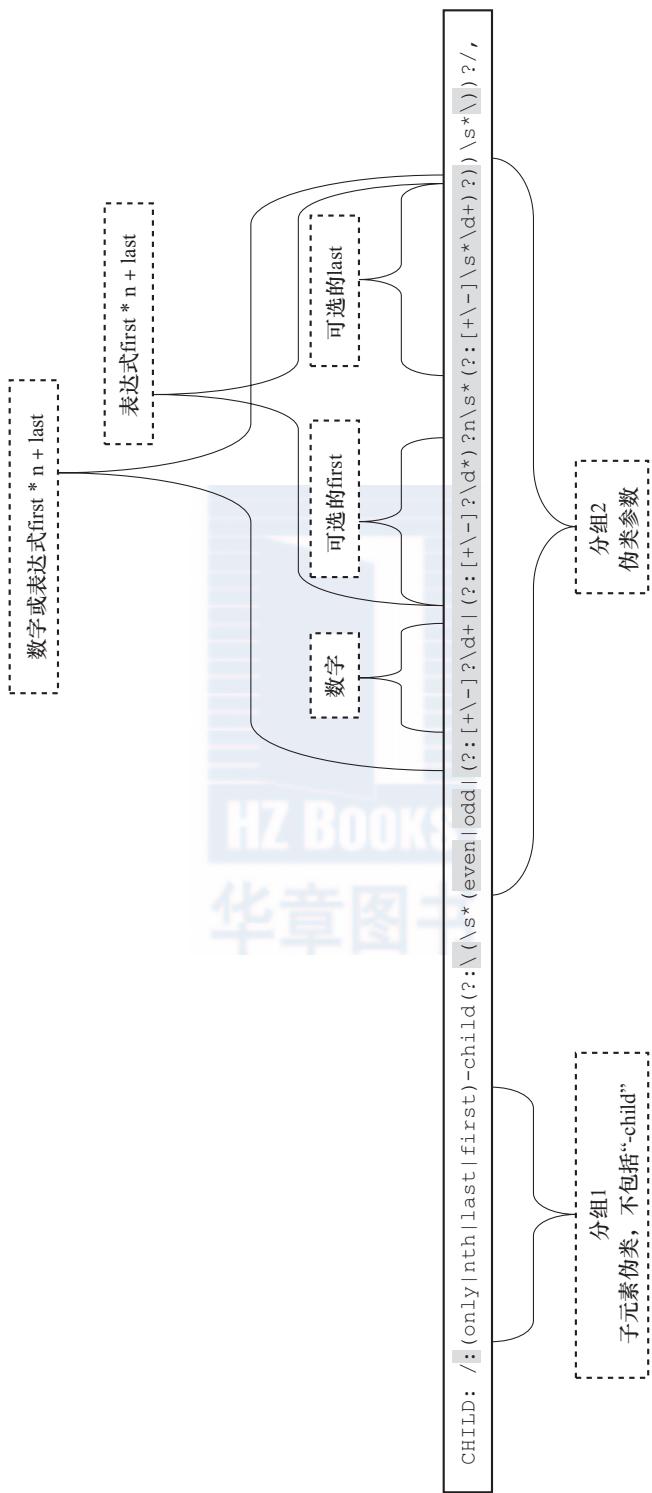


图 3-12 正则 Sizzle.selectors.match.CHILD

表 3-8 正则 Sizzle.selectors.match.CHILD

序号	测试用例	运行结果
1	CHILD.exec("p:first-child")	[":first-child", "first", undefined]
2	CHILD.exec("p:only-child")	[":only-child", "only", undefined]
3	CHILD.exec("option:nth-child")	[":nth-child", "nth", undefined]
4	CHILD.exec("option:nth-child(even)")	[":nth-child(even)", "nth", "even"]
5	CHILD.exec("option:nth-child(odd)")	[":nth-child(odd)", "nth", "odd"]
6	CHILD.exec("option:nth-child(1)")	[":nth-child(1)", "nth", "1"]
7	CHILD.exec("option:nth-child(+1)")	[":nth-child(+1)", "nth", "+1"]
8	CHILD.exec("option:nth-child(-1)")	[":nth-child(-1)", "nth", "-1"]
9	CHILD.exec("option:nth-child(0n+3)")	[":nth-child(0n+3)", "nth", "0n+3"]
10	CHILD.exec("option:nth-child(1n)")	[":nth-child(1n)", "nth", "1n"]
11	CHILD.exec("option:nth-child(n)")	[":nth-child(n)", "nth", "n"]
12	CHILD.exec("option:nth-child(+n)")	[":nth-child(+n)", "nth", "+n"]
13	CHILD.exec("option:nth-child(-1n+3)")	[":nth-child(-1n+3)", "nth", "-1n+3"]
14	CHILD.exec("option:nth-child(-n+3)")	[":nth-child(-n+3)", "nth", "-n+3"]
15	CHILD.exec("option:nth-child(-1n+3)")	[":nth-child(-1n+3)", "nth", "-1n+3"]
16	CHILD.exec("option:nth-child(2n)")	[":nth-child(2n)", "nth", "2n"]
17	CHILD.exec("option:nth-child(2n+1)")	[":nth-child(2n+1)", "nth", "2n+1"]
18	CHILD.exec("option:nth-child(2n+1)")	[":nth-child(2n+1)", "nth", "2n+1"]
19	CHILD.exec("option:nth-child(+2n+1)")	[":nth-child(+2n+1)", "nth", "+2n+1"]
20	CHILD.exec("option:nth-child(3n)")	[":nth-child(3n)", "nth", "3n"]
21	CHILD.exec("option:nth-child(3n+0)")	[":nth-child(3n+0)", "nth", "3n+0"]
22	CHILD.exec("option:nth-child(3n+1)")	[":nth-child(3n+1)", "nth", "3n+1"]
23	CHILD.exec("option:nth-child(3n-0)")	[":nth-child(3n-0)", "nth", "3n-0"]
24	CHILD.exec("option:nth-child(3n-1)")	[":nth-child(3n-1)", "nth", "3n-1"]

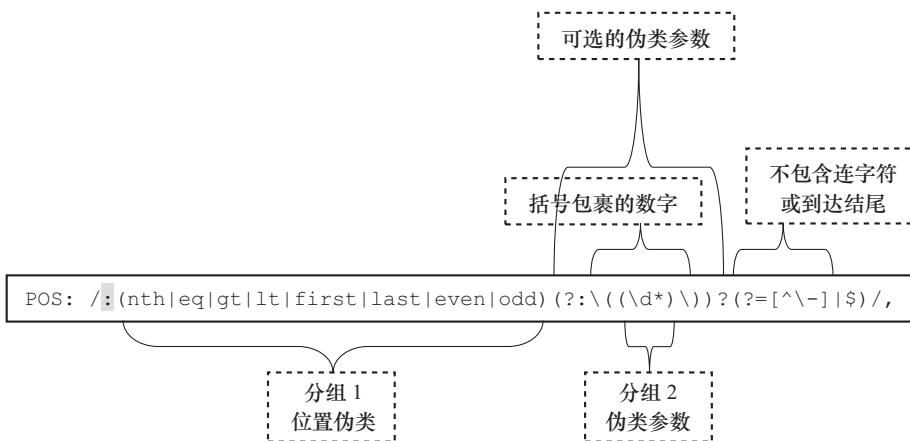


图 3-13 正则 Sizzle.selectors.match.POS

表 3-9 正则 Sizzle.selectors.match.POS

序号	测试用例	运行结果
1	POS.exec("p:nth(1)")	[":nth(1)", "nth", "1"]
2	POS.exec("p:eq(2)")	[":eq(2)", "eq", "2"]
3	POS.exec("p:gt(3)")	[":gt(3)", "gt", "3"]
4	POS.exec("p:lt(4)")	[":lt(4)", "lt", "4"]
5	POS.exec("p:first")	[":first", "first", undefined]
6	POS.exec("p:last")	[":last", "last", undefined]
7	POS.exec("p:even")	[":even", "even", undefined]
8	POS.exec("p:odd")	[":odd", "odd", undefined]

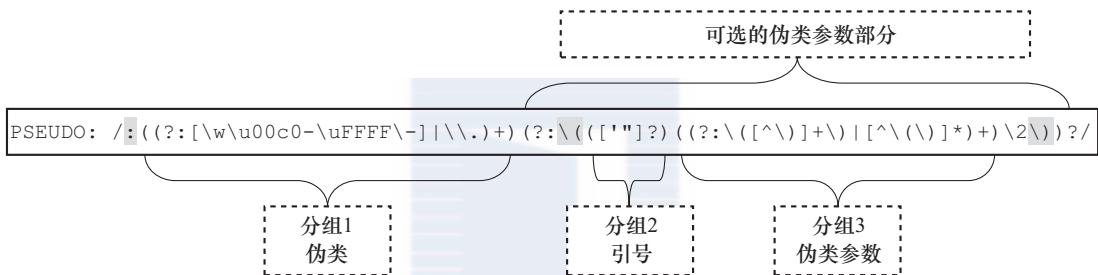


图 3-14 正则 Sizzle.selectors.match.PSEUDO

表 3-10 正则 Sizzle.selectors.match.PSEUDO

序号	测试用例	运行结果
1	PSEUDO.exec("p:has(a)")	[":has(a)", "has", "", "a"]
2	PSEUDO.exec("a:contains(Google)")	[":contains(Google)", "contains", "", "Google"]
3	PSEUDO.exec("input:focus")	[":focus", "focus", undefined, undefined]
4	PSEUDO.exec(":input")	[":input", "input", undefined, undefined]
5	PSEUDO.exec(".radio")	[":radio", "radio", undefined, undefined]
6	PSEUDO.exec(":checkbox")	[":checkbox", "checkbox", undefined, undefined]
7	PSEUDO.exec(":text")	[":text", "text", undefined, undefined]
8	PSEUDO.exec(":radio:checked")	[":radio", "radio", undefined, undefined]
9	PSEUDO.exec(":checkbox:checked")	[":checkbox", "checkbox", undefined, undefined]
10	PSEUDO.exec("option:selected")	[":selected", "selected", undefined, undefined]
11	PSEUDO.exec(":header")	[":header", "header", undefined, undefined]
12	PSEUDO.exec(":empty")	[":empty", "empty", undefined, undefined]
13	PSEUDO.exec(":parent")	[":parent", "parent", undefined, undefined]
14	PSEUDO.exec(":hidden")	[":hidden", "hidden", undefined, undefined]
15	PSEUDO.exec(":visible")	[":visible", "visible", undefined, undefined]

3.9.3 Sizzle.selectors.find

对象 Sizzle.selectors.find 中定义了 ID、CLASS、NAME、TAG 所对应的查找函数，称为“查找函数集”。其中，CLASS 需要浏览器支持方法 getElementsByClassName()。

查找函数会返回元素集合或 undefined，内部通过调用相应的原生方法来查找元素，如表 3-11 所示。查找函数调用原生方法前会检查上下文是否支持原生方法。

表 3-11 查找函数集 Sizzle.selectors.find

序号	类型	原生方法	说明
1	ID	getElementById()	查找拥有指定 id 的第一个元素
2	CLASS	getElementsByClassName()	查获拥有指定类样式的元素集合
3	NAME	getElementsByName()	查获拥有指定 name 的元素集合
4	TAG	getElementsByTagName()	查找拥有指定标签名的元素集合

1. ID

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4340     find: {
4341         ID: function( match, context, isXML ) {
4342             if ( typeof context.getElementById !== "undefined" && !isXML ) {
4343                 var m = context.getElementById(match[1]);
4344                 // Check parentNode to catch when BlackBerry 4.6 returns
4345                 // nodes that are no longer in the document #6963
4346                 return m && m.parentNode ? [m] : [];
4347             }
4348         },
4370     },
4749 };
```

2. CLASS

相关代码如下所示：

```
5139 (function() {
    // 测试浏览器是否支持方法 getElementsByClassName()
    // 如果不支持，则不做任何事情

    // 如果当前浏览器支持方法 getElementsByClassName()
    5157     Expr.order.splice(1, 0, "CLASS");
    5158     Expr.find.CLASS = function( match, context, isXML ) {
    5159         if ( typeof context.getElementsByClassName !== "undefined" && !isXML ) {
    5160             return context.getElementsByClassName(match[1]);
    5161         }
    5162     };
5166 })();
```

3. NAME

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4340     find: {
4350         NAME: function( match, context ) {
4351             if ( typeof context.getElementsByName !== "undefined" ) {
4352                 var ret = [],
4353                     results = context.getElementsByName( match[1] );
4354
4355                 for ( var i = 0, l = results.length; i < l; i++ ) {
4356                     if ( results[i].getAttribute("name") === match[1] ) {
4357                         ret.push( results[i] );
4358                     }
4359                 }
4360
4361                 return ret.length === 0 ? null : ret;
4362             }
4363         },
4364
4370     },
4749 };

```

4. TAG

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4340     find: {
4350         TAG: function( match, context ) {
4351             if ( typeof context.getElementsByTagName !== "undefined" ) {
4352                 return context.getElementsByTagName( match[1] );
4353             }
4354         }
4368
4370     },
4749 };

```

3.9.4 Sizzle.selectors.preFilter

对象 Sizzle.selectors.preFilter 中定义了类型 CLASS、ID、TAG、CHILD、ATTR、PSEUDO、POS 所对应的预过滤函数，称为“预过滤函数集”。

在方法 Sizzle.filter(expr, set, inplace, not) 中，预过滤函数在过滤函数 Sizzle.selectors.filter[type] 之前被调用，见图 3-6。调用预过滤函数时的参数格式为：

Sizzle.selectors.preFilter[type](正则匹配结果 match, 元素集合 curLoop, 是否缩小元

素集合 `inplace`, 新集合 `result`, 是否取反 `not`, `isXML`)

预过滤函数用于在过滤函数之前修正与过滤操作相关的参数, 每种类型的预过滤函数其修正行为如表 3-12 所示。

表 3-12 预过滤函数集 Sizzle.selectors.preFilter

序号	类型	修正行为	序号	类型	修正行为
1	CLASS	过滤不匹配元素, 或缩小元素集合	5	ATTR	修正属性名和属性值
2	ID	过滤转义反斜杠	6	PSEUDO	处理 :not(selector) 的伪类参数
3	TAG	过滤转义反斜杠, 转为小写	7	POS	修正位置伪类的参数下标
4	CHILD	格式化子元素伪类参数			

预过滤函数有 3 种返回值, 对应的含义如表 3-13 所示。

表 3-13 预过滤函数的返回值

序号	返回值	说明
1	false	已经执行过滤, 或已经缩小候选集, 不需要再执行过滤函数, 例如, CLASS
2	true	需要继续执行其他的预过滤函数, 尚不到执行过滤函数的时候, 例如, 在 PSEUDO 预过滤函数中遇到 POS、CHILD 时
3	其他	可以调用对应的过滤函数

对象 Sizzle.selectors.preFilter 的总体源码结构如下所示:

```
var Expr = Sizzle.selectors = {
    preFilter: {
        CLASS: function( match, curLoop, inplace, result, not, isXML ) { ... },
        ID: function( match ) { ... },
        TAG: function( match, curLoop ) { ... },
        CHILD: function( match ) { ... },
        ATTR: function( match, curLoop, inplace, result, not, isXML ) { ... },
        PSEUDO: function( match, curLoop, inplace, result, not ) { ... },
        POS: function( match ) { ... }
    },
};
```

下面对其中的预过滤函数逐个进行介绍和分析。

1. CLASS

类样式预过滤函数 Sizzle.selectors.preFilter.CLASS(`match`, `curLoop`, `inplace`, `result`, `not`, `isXML`) 负责检查元素集合中的每个元素是否含有指定的类样式。如果参数 `inplace` 为 true, 则将不匹配的元素替换为 false; 如果参数 `inplace` 不是 true, 则将匹配元素放入元素集合 `result` 中, 以此来不断地缩小元素集合。关于正则 Sizzle.selectors.match/leftMatch.CLASS 的说明请参见 3.9.2 节。

相关代码如下所示:

```
3866     rBackslash = /\//g,
4221 var Expr = Sizzle.selectors = {
```

```

4371     preFilter: {
4372         CLASS: function( match, curLoop, inplace, result, not, isXML ) {
4373             match = " " + match[1].replace( rBackslash, "" ) + " ";
4374
4375             if ( isXML ) {
4376                 return match;
4377             }
4378
4379             for ( var i = 0, elem; (elem = curLoop[i]) != null; i++ ) {
4380                 if ( elem ) {
4381                     if ( not ^ (elem.className && (" " + elem.className +
" ")).replace(/[\t\n\r]/g, " ").indexOf(match) >= 0 ) {
4382                         if ( !inplace ) {
4383                             result.push( elem );
4384                         }
4385
4386                         } else if ( inplace ) {
4387                             curLoop[i] = false;
4388                         }
4389                     }
4390                 }
4391
4392             return false;
4393         },
4394     },
4395     },
4396 },
4397 },
4398 },
4399 };

```

第 4373 行：检查类样式的技巧是在前后加空格，然后用字符串方法 `indexOf()` 进行判断。

第 4379 ~ 4390 行：遍历元素集合 `curLoop`，检测每个元素是否含有指定名称的类样式；如果参数 `not` 不是 `true`，则保留匹配元素，并排除不匹配元素；如果参数 `not` 是 `true`，则保留不匹配元素，排除匹配元素。如果参数 `inplace` 为 `true`，则将不匹配的元素替换为 `false`；如果参数 `inplace` 不是 `true`，则不修改元素集合 `curLoop`，而是将匹配元素放入元素集合 `result` 中，以此来不断地缩小元素集合。

第 4381 ~ 4388 行：这段 `if-else-if` 语句块的逻辑有些绕，可以这样理解：`if` 代码块表示的是过滤时通过了的情况，`else-if` 语句块表示的是过滤时未通过的情况。

第 4392 行：`CLASS` 预过滤函数总是返回 `false`，表示已经执行过滤，或已缩小候选集，不需要再执行 `CLASS` 过滤函数。

2. ID

`ID` 预过滤函数 `Sizzle.selectors.preFilter.ID(match)` 负责过滤转义反斜杠，从匹配结果 `match` 中提取并返回 `id` 值。关于正则 `Sizzle.selectors.match/leftMatch.ID` 的具体说明请参见 3.9.2 节。

相关代码如下所示：

```

3866     rBackslash = /\\\/g,
4221 var Expr = Sizzle.selectors = {

```

```

4371     preFilter: {
4395         ID: function( match ) {
4396             return match[1].replace( rBackslash, "" );
4397         },
4475     },
4749 };

```

3. TAG

标签预过滤函数 Sizzle.selectors.preFilter.TAG(match, curLoop) 负责过滤转义反斜杠，转换为小写，从匹配结果 match 中提取并返回标签名。关于正则 Sizzle.selectors.match/leftMatch.TAG 的具体说明参见 3.9.2 节。

相关代码如下所示：

```

3866     rBackslash = /\$/g,
4221 var Expr = Sizzle.selectors = {
4371     preFilter: {
4399         TAG: function( match, curLoop ) {
4400             return match[1].replace( rBackslash, "" ).toLowerCase();
4401         },
4475     },
4749 };

```

4. CHILD

子元素伪类预过滤函数 Sizzle.selectors.preFilter.CHILD(match) 负责将伪类 :nth-child(index/even/odd/equation) 的参数格式化为 first*n + last，例如，将 odd 格式化为 2n+1。关于正则 Sizzle.selectors.match/leftMatch.CHILD 的具体说明请参见 3.9.2 节。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4371     preFilter: {
4403         CHILD: function( match ) {
4404             if ( match[1] === "nth" ) {
4405                 if ( !match[2] ) {
4406                     Sizzle.error( match[0] );
4407                 }
4408                 match[2] = match[2].replace( /^\\+|\\s*/g, '' );
4409                 //parse equations like 'even', 'odd', '5', '2n', '3n+2', '4n-1',
4410                 //'-n+6'
4411             }
4412         }
4413     }
4414 };

```

```

4412     var test = /(-?)(\d*)(?:n([+-]?\d*))?/.exec(
4413         match[2] === "even" && "2n" ||
4414         match[2] === "odd" && "2n+1" ||
4415         !/\D/.test( match[2] ) && "0n+" + match[2] ||
4416         match[2]);
4417
4418     // calculate the numbers (first)n+(last) including if they are
4419     negative
4420     match[2] = (test[1] + (test[2] || 1)) - 0;
4421     match[3] = test[3] - 0;
4422 }
4423
4424 // TODO: Move to normal caching system
4425 match[0] = done++;
4426
4427     return match;
4428 },
4429 },
4430 );

```

第 4409 行：替换伪类开头的加号和包含的空格，例如，`:nth-child(+1)`→`:nth-child(1)`、`:nth-child(2n + 1)`→`:nth-child(2n+1)`。

第 4412 ~ 4414 行：将伪类参数统一格式化为 `first*n + last`，例如，`even`→`2n`、`odd`→`2n+1`、`数字`→`0n+ 数字`。正则 `/(-?)(\d*)(?:n([+-]?\d*))?/` 含有 3 个分组：负号、`first` 部分、`last` 部分。

第 4417 ~ 4418 行：计算 `first` 部分和 `last` 部分。注意减 0 是为了将字符串强制转换为数值。

第 4425 行：为本次过滤分配一个唯一的标识，用于优化过滤过程，请参见 3.9.7 节对子元素伪类过滤函数 `Sizzle.selectors.filter.CHILD(elem, match)` 的介绍和分析。

5. ATTR

属性预过滤函数 `Sizzle.selectors.preFilter.ATTR(match, curLoop, inplace, result, not, isXML)` 负责修正匹配结果 `match` 中的属性名和属性值。关于正则 `Sizzle.selectors.match/leftMatch.ATTR` 的具体说明请参见 3.9.2 节。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4222     preFilter: {
4223         ATTR: function( match, curLoop, inplace, result, not, isXML ) {
4224             var name = match[1] = match[1].replace( rBackslash, "" );
4225
4226             if ( !isXML && Expr.attrMap[name] ) {
4227                 match[1] = Expr.attrMap[name];

```

```

4435      }
4436
4437      // Handle if an un-quoted value was used
4438      match[4] = ( match[4] || match[5] || "" ).replace( rBackslash, "" );
4439
4440      if ( match[2] === "~=" ) {
4441          match[4] = " " + match[4] + " ";
4442      }
4443
4444      return match;
4445  },
4475  },
4749 };

```

第 4431 ~ 4435 行：修正属性名。删除转义反斜杠，修正某些特殊属性名。

第 4438 ~ 4442 行：修正属性值。合并分组 4 和分组 5 的值，删除转义反斜杠。当属性表达式的属性值有引号时，属性值存储在 match[4]，否则存储在 match[5]。如果等号部分是 ~=，表示是单词匹配，则在属性值前后加空格；在过滤函数中，对于 ~=，会在元素的属性值前后加空格，然后用字符串方法 indexOf() 检查。

6. PSEUDO

伪类预过滤函数 Sizzle.selectors.preFilter.PSEUDO(match, curLoop, inplace, result, not) 主要负责处理伪类表达式是 :not(selector) 的情况，该函数会将匹配结果 match 中的分组 3（即伪类参数 selector）替换为与之匹配的元素集合；对于位置伪类和子元素伪类，则返回 true，继续执行各自对应的预过滤函数。关于正则 Sizzle.selectors.match/leftMatch.PSEUDO 的具体说明请参见 3.9.2 节。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4371     preFilter: {
4447         PSEUDO: function( match, curLoop, inplace, result, not ) {
4448             if ( match[1] === "not" ) {
4449                 // If we're dealing with a complex expression, or a simple
one
4450                 if ( ( chunker.exec(match[3]) || "" ).length > 1 || /^\\w/.test(match[3]) ) {
4451                     match[3] = Sizzle(match[3], null, null, curLoop);
4452
4453                 } else {
4454                     var ret = Sizzle.filter(match[3], curLoop, inplace, true ^
not);
4455
4456                     if ( !inplace ) {
4457                         result.push.apply( result, ret );
4458                     }
4459
4460                 return false;

```

```

4461         }
4462     } else if ( Expr.match.POS.test( match[0] ) || Expr.match.
CHILD.test( match[0] ) ) {
4464         return true;
4465     }
4466
4467     return match;
4468 },
4475 },
4749 };

```

第 4447 行：参数 match 是正则 Sizzle.selectors.match.PSEUDO 匹配块表达式的结果，含有 3 个分组：伪类、引号、伪类参数。

第 4448 ~ 4461 行：如果伪类是 :not(selector)，则将匹配结果 match 中的分组 3（即伪类参数 selector）替换为与之其匹配的元素集合。在对应的过滤函数中，会筛选出不在分组 3 中的元素。

第 4463 ~ 4465 行：如果是位置伪类 POS 或子元素伪类 CHILD，则返回 true，表示仍然需要继续执行各自所对应的预过滤函数。注意，位置伪类 POS 和子元素伪类 CHILD 有着自己的预过滤函数。

7. POS

位置伪类预过滤函数 Sizzle.selectors.preFilter.POS(match) 负责在匹配结果 match 的头部插入一个新元素 true，使得匹配结果 match 中位置伪类参数的下标变为了 3，从而与伪类的匹配结果保持一致。关于正则 Sizzle.selectors.match/leftMatch.POS/PSEUDO 的具体说明请参见 3.9.2 节。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4371     preFilter: {
4470         POS: function( match ) {
4471             match.unshift( true );
4472
4473             return match;
4474         }
4475     },
4749 };

```

3.9.5 Sizzle.selectors.filters

对象 Sizzle.selectors.filters 中定义了一组伪类和对应的伪类过滤函数，称为“伪类过滤函数集”。支持的伪类有：:enabled、:disabled、:checked、:selected、:parent、:empty、:has、:header、:text、:radio、:checkbox、:file、:password、:submit、:image、:reset、:button、:input、:focus。方法调用链为：

Sizzle.filter() → Sizzle.selectors.filter.PSEUDO() → Sizzle.selectors.filters，如图3-1所示。

伪类过滤函数负责检查元素是否匹配伪类，返回一个布尔值，其参数格式为：

```
Sizzle.selectors.filters[ 伪类 ]( 元素 , 序号 , 正则匹配结果 , 元素集合 );
// 正则匹配结果是正则 Sizzle.selectors.match.PSEUDO 匹配块选择器表达式的结果，含有 3 个分组：伪类、引号、伪类参数
```

相关代码如下所示，为了方便解释，代码中增加了示例和注释：

```
4221 var Expr = Sizzle.selectors = {
4477     filters: {
4478         ':enabled': // $(':enabled') 匹配所有可用元素（未禁用的，不隐藏的）
4479         enabled: function( elem ) {
4480             return elem.disabled === false && elem.type !== "hidden";
4481         },
4482         ':disabled': // $(':disabled') 匹配所有不可用元素（禁用的）
4483         disabled: function( elem ) {
4484             return elem.disabled === true;
4485         },
4486         ':checked': // $(':checked') 匹配所有选中的被选中元素，包括复选框、单选按钮，不包括 option
元素
4487         checked: function( elem ) {
4488             return elem.checked === true;
4489         },
4490         ':selected': // $(':selected') 匹配所有选中的 option 元素
4491         selected: function( elem ) {
4492             // Accessing this property makes selected-by-default
4493             // options in Safari work properly
4494             if ( elem.parentNode ) {
4495                 elem.parentNode.selectedIndex;
4496             }
4497             return elem.selected === true;
4498         },
4499         ':parent': // $(':parent') 匹配所有含有子元素或文本的元素
4500         parent: function( elem ) {
4501             return !!elem.firstChild;
4502         },
4503         ':empty': // $(':empty') 匹配所有不包含子元素或者文本的空元素
4504         empty: function( elem ) {
4505             return !elem.firstChild;
4506         },
4507         ':has(selector)': // $(':has(selector)') 匹配含有选择器所匹配元素的元素
4508         has: function( elem, i, match ) {
4509             return !Sizzle( match[3], elem ).length;
4510         },
4511         ':header': // $(':header') 匹配如 h1、h2、h3 之类的标题元素
4512         header: function( elem ) {
4513             return (/h\d/i).test( elem.nodeName );
4514         },
4515         ':text': // $(':text') 匹配所有单行文本框
4516         text: function( elem ) {
4517             var attr = elem.getAttribute( "type" ), type = elem.type;
4518             // IE6 and 7 will map elem.type to 'text' for new HTML5 types (search, etc)
```

```

4519           //use getAttribute instead to test this case
4520           return elem.nodeName.toLowerCase() === "input" && "text" ===
type && ( attr === type || attr === null );
4521           },
4522           //$(':radio') 匹配所有单选按钮
4523           radio: function( elem ) {
4524               return elem.nodeName.toLowerCase() === "input" && "radio" ===
elem.type;
4525           },
4526           //$(':checkbox') 匹配所有复选框
4527           checkbox: function( elem ) {
4528               return elem.nodeName.toLowerCase() === "input" && "checkbox" ===
elem.type;
4529           },
4530           //$(':file') 匹配所有文件域
4531           file: function( elem ) {
4532               return elem.nodeName.toLowerCase() === "input" && "file" ===
elem.type;
4533           },
4534           //$(':password') 匹配所有密码框
4535           password: function( elem ) {
4536               return elem.nodeName.toLowerCase() === "input" && "password" ===
elem.type;
4537           },
4538           //$(':submit') 匹配所有提交按钮
4539           submit: function( elem ) {
4540               var name = elem.nodeName.toLowerCase();
4541               return (name === "input" || name === "button") && "submit" ===
elem.type;
4542           },
4543           //$(':image') 匹配所有图像域
4544           image: function( elem ) {
4545               return elem.nodeName.toLowerCase() === "input" && "image" ===
elem.type;
4546           },
4547           //$(':reset') 匹配所有重置按钮
4548           reset: function( elem ) {
4549               var name = elem.nodeName.toLowerCase();
4550               return (name === "input" || name === "button") && "reset" ===
elem.type;
4551           },
4552           //$(':button') 匹配所有按钮
4553           button: function( elem ) {
4554               var name = elem.nodeName.toLowerCase();
4555               return name === "input" && "button" === elem.type || name ===
"button";
4556           },
4557           //$(':input') 匹配所有 input、textarea、select、button 元素
4558           input: function( elem ) {
4559               return (/input|select|textarea|button/i).test( elem.nodeName );
4560           },
4561           //$(':focus') 匹配当前焦点元素
4562           focus: function( elem ) {
4563               return elem === elem.ownerDocument.activeElement;
4564           }

```

```
4565      },
4749  };
```

3.9.6 Sizzle.selectors.setFilters

对象 Sizzle.selectors.setFilters 中定义了一组位置伪类和对应的伪类过滤函数，称为“位置伪类过滤函数集”。支持的位置伪类有：:first、:last、:even、:odd、:lt(index)、:gt(index)、:nth(index)、:eq(index)。方法调用链为：Sizzle.filter() → Sizzle.selectors.filter.POS() → Sizzle.selectors.setFilters，如图 3-1 所示。

位置伪类过滤函数通过比较下标来确定元素在集合中的位置，返回一个布尔值，其参数格式为：

```
Sizzle.selectors.setFilters[ 位置伪类 ]( 元素 , 下标 , 正则匹配结果 , 元素集合 );
// 正则匹配结果是正则 Sizzle.selectors.match.POS 匹配块选择器表达式的结果，含有 2 个分组：
位置伪类、位置伪类参数
```

相关代码如下所示，为了方便解释，代码中增加了示例和注释：

```
4221 var Expr = Sizzle.selectors = {
4566   setFilters: {
4567     // $(':first') 匹配找到的第一个元素
4568     first: function( elem, i ) {
4569       return i === 0;
4570     },
4571     // $(':last') 匹配找到的最后一个元素
4572     last: function( elem, i, match, array ) {
4573       return i === array.length - 1;
4574     },
4575     // $(':even') 匹配所有下标为偶数的元素，从 0 开始计数
4576     even: function( elem, i ) {
4577       return i % 2 === 0;
4578     },
4579     // $(':odd') 匹配所有下标为奇数的元素，从 0 开始计数
4580     odd: function( elem, i ) {
4581       return i % 2 === 1;
4582     },
4583     // $(':lt(index)') 匹配所有小于指定下标的元素
4584     lt: function( elem, i, match ) {
4585       return i < match[3] - 0;
4586     },
4587     // $(':gt(index)') 匹配所有大于指定下标的元素
4588     gt: function( elem, i, match ) {
4589       return i > match[3] - 0;
4590     },
4591     // $(':nth(index)') 匹配一个指定下标的元素，从 0 开始计数
4592     nth: function( elem, i, match ) {
4593       return match[3] - 0 === i;
4594     },
4595     // $(':eq(index)') 匹配一个指定下标的元素，从 0 开始计数
4596     eq: function( elem, i, match ) {
4597       return match[3] - 0 === i;
```

```

4597         }
4598     },
4749 };

```

3.9.7 Sizzle.selectors.filter

对象 Sizzle.selectors.filter 中定义了类型 PSEUDO、CHILD、ID、TAG、CLASS、ATTR、POS 所对应的过滤函数，称为“过滤函数集”。方法调用链为：Sizzle.filter() → Sizzle.selectors.filter[type]，如图 3-1 和图 3-6 所示。

过滤函数负责检查元素是否匹配过滤表达式，返回一个布尔值，其参数格式为：

```

Sizzle.selectors.filter[ 类型 ]( 元素 , 正则匹配结果或过滤表达式 , 下标 , 元素集合 )
// 正则匹配结果指 Sizzle.selectors.match 中对应的正则匹配块表达式的结果
// 过滤表达式指经过 Sizzle.selectors.preFilter 处理后的块表达式

```

Sizzle.selectors.filter 的总体源码结构如下所示：

```

var Expr = Sizzle.selectors = {
  filter: {
    PSEUDO: function( elem, match, i, array ) { ... },
    CHILD: function( elem, match ) { ... },
    ID: function( elem, match ) { ... },
    TAG: function( elem, match ) { ... },
    CLASS: function( elem, match ) { ... },
    ATTR: function( elem, match ) { ... },
    POS: function( elem, match, i, array ) { ... }
  }
};

```

下面对其中的过滤函数逐个进行介绍和分析。

1. PSEUDO

伪类过滤函数 Sizzle.selectors.filter.PSEUDO(elem, match, i, array) 用于检查元素是否匹配伪类。大部分检查通过调用伪类过滤函数集 Sizzle.selectors.filters 中对应的伪类过滤函数来实现，对于伪类 :contains(text)、:not(selector) 则做特殊处理。具体请参见 3.9.5 节。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {

4599   filter: {
4600     PSEUDO: function( elem, match, i, array ) {
4601       var name = match[1],
4602           filter = Expr.filters[ name ];
4603
4604       if ( filter ) {
4605         return filter( elem, i, match, array );
4606
4607       } else if ( name === "contains" ) {
4608         return (elem.textContent || elem.innerText || getText([ elem ]) || "").
indexOf(match[3]) >= 0;

```

```

4609
4610      } else if ( name === "not" ) {
4611          var not = match[3];
4612
4613          for ( var j = 0, l = not.length; j < l; j++ ) {
4614              if ( not[j] === elem ) {
4615                  return false;
4616              }
4617          }
4618
4619          return true;
4620
4621      } else {
4622          Sizzle.error( name );
4623      }
4624  },
4748
4749 };

```

第 4600 行：参数 match 是正则 Sizzle.selectors.match.PSEUDO 匹配块表达式的结果，含有 3 个分组：伪类、引号、伪类参数。

第 4602 ~ 4605 行：如果在伪类过滤函数集 Sizzle.selectors.filters 中存在对应的伪类过滤函数，则调用它来检查元素是否匹配伪类。

第 4607 ~ 4608 行：伪类 :contains(text) 用于匹配包含指定文本的所有元素。如果伪类是 :contains(text)，则先取出当前元素的文本内容，然后调用字符串方法 indexOf() 检查是否含有指定的文本。

第 4610 ~ 4619 行：伪类 :not(selector) 用于匹配与指定选择器不匹配的所有元素。如果伪类是 :not(selector)，则检查当前元素是否与 match[3] 中的某个元素相等，如果相等则返回 false，否则返回 true。在预过滤函数 Sizzle.selectors.preFilter.PSEUDO 中，对于伪类 :not(selector)，会将 match[3] 替换为其匹配的元素集合。

第 4621 ~ 4623 行：对于不支持的伪类，一律调用方法 Sizzle.error(msg) 抛出语法错误。方法 Sizzle.error(msg) 请参见 3.10.4 节。

2. CHILD

子元素伪类过滤函数 Sizzle.selectors.filter.CHILD(elem, match) 用于检查元素是否匹配子元素伪类。支持的子元素伪类如表 3-14 所示。

表 3-14 子元素伪类

序号	子元素伪类	说 明
1	:nth-child(index/even/odd/equation)	匹配父元素下的第 N 个子元素或奇偶元素
2	:first-child	匹配父元素的第一个子元素
3	:last-child	匹配父元素的最后一个子元素
4	:only-child	如果某个元素是父元素的唯一子元素，则匹配；如果父元素还含有多个子元素，则不匹配

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4599     filter: {
4626         CHILD: function( elem, match ) {
4627             var first, last,
4628                 doneName, parent, cache,
4629                 count, diff,
4630                 type = match[1],
4631                 node = elem;
4632
4633             switch ( type ) {
4634                 case "only":
4635                 case "first":
4636                     while ( (node = node.previousSibling) ) {
4637                         if ( node.nodeType === 1 ) {
4638                             return false;
4639                         }
4640                     }
4641
4642                     if ( type === "first" ) {
4643                         return true;
4644                     }
4645
4646                     node = elem;
4647
4648                 case "last":
4649                     while ( (node = node.nextSibling) ) {
4650                         if ( node.nodeType === 1 ) {
4651                             return false;
4652                         }
4653                     }
4654
4655                     return true;
4656
4657                 case "nth":
4658                     first = match[2];
4659                     last = match[3];
4660
4661                     if ( first === 1 && last === 0 ) {
4662                         return true;
4663                     }
4664
4665                     doneName = match[0];
4666                     parent = elem.parentNode;
4667
4668                     if ( parent && (parent[ expando ] !== doneName || !elem.
nodeIndex) ) {
4669                         count = 0;
4670
4671                         for ( node = parent.firstChild; node; node = node.
nextSibling ) {
4672                             if ( node.nodeType === 1 ) {
4673                                 if ( count === last ) {
4674                                     return true;
4675                                 }
4676                             }
4677                         }
4678
4679                     }
4680
4681                     if ( count === last ) {
4682                         return true;
4683                     }
4684
4685                 }
4686             }
4687         }
4688     }
4689 }
```

```

4673                     node.nodeIndex = ++count;
4674                 }
4675             }
4676         }
4677         parent[ expando ] = doneName;
4678     }
4679
4680     diff = elem.nodeIndex - last;
4681
4682     if ( first === 0 ) {
4683         return diff === 0;
4684     }
4685     } else {
4686         return ( diff % first === 0 && diff / first >= 0 );
4687     }
4688 }
4689 },
4748 }
4749 };

```

第 4634 ~ 4655 行：如果伪类是 :only-child，则检查当前元素之前（previousSibling）和之后（nextSibling）是否有兄弟元素，如果有则返回 true，否则返回 false。注意这里的分支 only 是通过分支 first 和分支 last 实现的。

第 4635 ~ 4644 行：如果伪类是 :first-child，则检查当前元素之前（previousSibling）是否有兄弟元素，有则返回 false，没有则返回 true。

第 4648 ~ 4655 行：如果伪类是 :last-child，则检查当前元素之后（nextSibling）是否有兄弟元素，有则返回 false，没有则返回 true。

第 4657 ~ 4687 行：如果伪类是 :nth-child(index/even/odd/equation)，则检查当前元素的下标是否匹配伪类参数，检测公式为：

$$(\text{当前元素在其父元素中的下标位置} - \text{last}) \% \text{first} === 0$$

在预过滤函数 Sizzle.selectors.preFilter.CHILD 中已将伪类参数统一格式化为 first*n+last，例如，odd 格式化为 2n+1，其中，first 存储在 match[2] 中，last 存储在 match[3] 中。

第 4665 ~ 4678 行：找到当前元素的父元素，然后为每个子元素设置属性 nodeIndex，从而标识出每个子元素的下标位置。如果父元素未被本次过滤标识过，或当前元素未被标识过，才会为子元素设置属性 nodeIndex，以确保只会标识一次。

match[0] 是本次过滤的唯一标识，在执行子元素预过滤函数 Sizzle.selectors.preFilter.CHILD(match) 时被分配，具体请参见 3.9.4 节。

3. ID

ID 过滤函数 Sizzle.selectors.filter.ID(elem, match) 用于检查元素的属性 id 是否与指定的 id 相等。

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4599     filter: {
4691         ID: function( elem, match ) {
4692             return elem.nodeType === 1 && elem.getAttribute("id") === match;
4693         },
4748     }
4749 };
```

4. TAG

标签过滤函数 Sizzle.selectors.filter.TAG(elem, match) 用于检查元素的标签名 nodeName 是否与指定的标签名相等。

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4599     filter: {
4691         ID: function( elem, match ) {
4692             return elem.nodeType === 1 && elem.getAttribute("id") === match;
4693         },
4748     }
4749 };
```

5. CLASS

类样式过滤函数 Sizzle.selectors.filter.CLASS(elem, match) 用于检查元素的类样式 className 是否含有指定的类样式。检查技巧是在类样式前后加空格，然后判断字符串方法 indexOf() 的返回值。

相关代码如下所示：

```
4221 var Expr = Sizzle.selectors = {
4599     filter: {
4699         CLASS: function( elem, match ) {
4700             return (" " + (elem.className || elem.getAttribute("class")) + " ")
4701                 .indexOf( match ) > -1;
4702         },
4748     }
4749 };
```

6. ATTR

属性过滤函数 Sizzle.selectors.filter.ATTR(elem, match) 用于检查元素的属性是否匹配属

性表达式。支持的属性表达式如表 3-15 所示。

表 3-15 属性表达式

序号	属性表达式	说 明
1	[attribute]	匹配含有指定属性的元素
2	[attribute=value]	匹配含有指定属性，并且当前属性值等于指定值的元素
3	[attribute!=value]	匹配不包含指定属性，或者当前属性值不等于指定值的元素
4	[attribute^=value]	匹配含有指定属性，并且属性值以指定值开始的元素
5	[attribute\$=value]	匹配含有指定属性，并且当前属性值以指定值结束的元素
6	[attribute*=value]	匹配含有指定属性，并且当前属性值包含指定值的元素
7	[attribute =“value”]	匹配含有指定属性，并且当前属性值等于指定值，或者当前属性值以指定值开头，并且后跟一个连字符（-）的元素
8	[attribute~=“value”]	匹配含有指定属性，并且当前属性值含有指定单词的元素。单词之间用空格分隔

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4599     filter: {
4704         ATTR: function( elem, match ) {
4705             var name = match[1],
4706                 result = Sizzle.attr ?
4707                     Sizzle.attr( elem, name ) :
4708                     Expr.attrHandle[ name ] ?
4709                         Expr.attrHandle[ name ]( elem ) :
4710                         elem[ name ] != null ?
4711                             elem[ name ] :
4712                             elem.getAttribute( name ),
4713                         value = result + "",
4714                         type = match[2],
4715                         check = match[4];
4716
4717             return result == null ?
4718                 type === "!=" :
4719                 !type && Sizzle.attr ?
4720                     result != null :
4721                     type === "=" ?
4722                         value === check :
4723                         type === "*=" ?
4724                         value.indexOf(check) >= 0 :
4725                         type === "~=" ?
4726                         (" " + value + " ").indexOf(check) >= 0 :
4727                         !check ?
4728                         value && result !== false :
4729                         type === "!=" ?
4730                         value !== check :
4731                         type === "^=" ?
4732                         value.indexOf(check) === 0 :
4733                         type === "$=" ?
4734                         value.substr(value.length - check.length) === check :
4735                         type === "|=" ?

```

```

4736         value === check || value.substr(0, check.length + 1) === check + "-" :
4737         false;
4738     },
4748 }
4749 };

```

第 4705 行：变量 name 是指定的属性名。

第 4706 ~ 4712 行：变量 result 是元素的 HTML 属性值或 DOM 属性值。在 jQuery 中，因为 Sizzle.attr() 等价于 jQuery.attr()，因此总是返回 HTML 属性，所以变量 result 也总是 HTML 属性值；在独立使用 Sizzle 时，则是先尝试读取 DOM 属性值，如果不存在才会读取 HTML 属性值。

第 4713 ~ 4715 行：变量 value 是变量 result 字符串格式；变量 type 是属性表达式的等号部分，例如，=、!=；变量 check 是指定的属性值。

第 4717 ~ 4737 行：根据等号部分，采用不同的比较方式来检查元素是否匹配属性表达式。由于这段复合三元表达式太长太复杂，因此，下面将格式稍做调整并加上注释，以便于阅读理解：

```

// [name!=value] 不包含指定属性
return result == null ? type === "!=" :
// [name] 包含指定属性
!type && Sizzle.attr ? result != null :
// [name=check] 包含指定属性，属性值等于指定值
type === "=" ? value === check :
// [name*=check] 含有指定属性，属性值包含指定值
type === "*=" ? value.indexOf(check) >= 0 :
// [name~="value"] 含有指定属性，属性值含有指定单词
type === "~=" ? (" " + value + " ").indexOf(check) >= 0 :
// 如果没有指定值 check，只有指定属性值，并且属性值不是 false，才会返回 true
!check ? value && result !== false :
// 以下均有指定值 check
// [name!=check] 含有指定属性，属性值不等于指定值
type === "!=" ? value !== check :
// [name^=check] 含有指定属性，属性值以指定值开始
type === "^=" ? value.indexOf(check) === 0 :
// [name$=check] 含有指定属性，属性值以指定值结束
type === "$=" ? value.substr(value.length - check.length) === check :
// [name|=check] 含有指定属性，属性值等于指定值，或者以指定值开头，且后跟一个连字符 (-)
type === "|=" ? value === check || value.substr(0, check.length + 1) ===
check + "-" :
false;

```

7. POS

位置伪类过滤函数 Sizzle.selectors.filter.POS(elem, match, i, array) 用于检查元素是否匹配位置伪类，该函数通过调用位置伪类过滤函数集 Sizzle.selectors.setFilters 中对应的位置伪类过滤函数来实现，具体请参见 3.9.6 节。调用关系如图 3-1 所示。

相关代码如下所示：

```

4221 var Expr = Sizzle.selectors = {
4599     filter: {
4740         POS: function( elem, match, i, array ) {
4741             var name = match[2],
4742                 filter = Expr.setFilters[ name ];
4743
4744             if ( filter ) {
4745                 return filter( elem, i, match, array );
4746             }
4747         }
4748     }
4749 };

```

3.10 工具方法

3.10.1 Sizzle.uniqueSort(results)

工具方法 Sizzle.uniqueSort(results) 负责对元素集合中的元素按照出现在文档中的顺序进行排序，并删除重复元素。

相关代码如下所示：

```

4026 Sizzle.uniqueSort = function( results ) {
4027     if ( sortOrder ) {
4028         hasDuplicate = baseHasDuplicate;
4029         results.sort( sortOrder );
4030
4031         if ( hasDuplicate ) {
4032             for ( var i = 1; i < results.length; i++ ) {
4033                 if ( results[i] === results[ i - 1 ] ) {
4034                     results.splice( i--, 1 );
4035                 }
4036             }
4037         }
4038     }
4039
4040     return results;
4041 };

```

第 4029 行：调用数组方法 sort() 对数组中的元素进行排序。其中，sortOrder(a, b) 是比较函数，负责比较元素 a 和元素 b 在文档中的位置。如果比较函数 sortOrder(a, b) 遇到相等的元素，即重复元素，会设置变量 hasDuplicate 为 true。关于比较函数 sortOrder(a, b) 的具体说明请参见 3.10.2 节。

第 4031 ~ 4037 行：如果变量 hasDuplicate 为 true，表示存在重复元素，则遍历数组 results，比较相邻元素是否相等，如果相等则删除。

第 4028 行：开始排序和去重时，先设置变量 hasDuplicate 的默认值为变量 baseHasDuplicate，变量 baseHasDuplicate 指示了 JavaScript 引擎在排序时是否会进行优化。

相关代码如下所示：

```

3864     hasDuplicate = false,
3865     baseHasDuplicate = true,

3870 // Here we check if the JavaScript engine is using some sort of
3871 // optimization where it does not always call our comparision
3872 // function. If that is the case, discard the hasDuplicate value.
3873 // Thus far that includes Google Chrome.
3874 [0, 0].sort(function() {
3875     baseHasDuplicate = false;
3876     return 0;
3877 });

```

第 3874 ~ 3877 行：检查 JavaScript 引擎在排序时是否会进行优化。在早期的 Chrome 浏览器中，排序时如果遇到相等的元素，不会调用比较函数，新版本中已经取消了这一优化。如果遇到相等元素便不调用比较函数，此时变量 `baseHasDuplicate` 默认为 `true`，即只能假设数组中含有重复元素；如果遇到相等元素时仍然会调用比较函数，则变量 `baseHasDuplicate` 将被设置为 `false`，这种情况下需要在比较函数中判断是否含有重复元素。

读者可以访问 <http://bugs.jquery.com/ticket/5380> 查看该 bug 的描述。

3.10.2 sortOrder(a, b)

函数 `sortOrder(a, b)` 负责比较元素 `a` 和元素 `b` 在文档中的位置。如果元素 `a` 在元素 `b` 之前，则返回 `-1`；如果元素 `a` 在元素 `b` 之后，则返回 `1`；如果元素 `a` 与元素 `b` 相等，则返回 `0`。

函数 `sortOrder(a, b)` 通过调用原生方法 `compareDocumentPosition()` 或比较原生属性 `sourceIndex` 来实现。原生方法 `compareDocumentPosition()` 用于比较两个元素的文档位置；原生属性 `sourceIndex` 则返回元素在文档中的序号，返回值等于该元素在 `document.getElementsByTagName('*')` 返回的数组中的下标。更多信息请访问 http://www.quirksmode.org/dom/w3c_core.html。

函数 `sortOrder(a, b)` 执行的 3 个关键步骤如下：

- 1) 如果浏览器支持原生方法 `compareDocumentPosition()`，则调用该方法比较元素位置。
- 2) 如果浏览器支持原生属性 `sourceIndex`，则用该属性比较元素位置。
- 3) 否则比较祖先元素的文档位置。

下面来看看该函数的源码实现。

1. 浏览器支持原生方法 `compareDocumentPosition()` 的情况

如果浏览器支持原生方法 `compareDocumentPosition()`，则调用该方法比较元素位置。相关代码如下所示：

```

4805 var sortOrder, siblingCheck;
4806
4807 if ( document.documentElement.compareDocumentPosition ) {
4808     sortOrder = function( a, b ) {
4809         if ( a === b ) {
4810             hasDuplicate = true;
4811             return 0;
4812         }

```

```

4813
4814     if ( !a.compareDocumentPosition || !b.compareDocumentPosition ) {
4815         return a.compareDocumentPosition ? -1 : 1;
4816     }
4817
4818     return a.compareDocumentPosition(b) & 4 ? -1 : 1;
4819 };
4820

```

2. 浏览器支持原生属性 sourceIndex 的情况

如果浏览器支持原生属性 sourceIndex，则用该属性比较元素位置。

相关代码如下所示：

```

4821 } else {
4822     sortOrder = function( a, b ) {
4823         // The nodes are identical, we can exit early
4824         if ( a === b ) {
4825             hasDuplicate = true;
4826             return 0;
4827
4828         // Fallback to using sourceIndex (in IE) if it's available on both nodes
4829         } else if ( a.sourceIndex && b.sourceIndex ) {
4830             return a.sourceIndex - b.sourceIndex;
4831         }
4832     }

```

3. 否则比较祖先元素的文档位置

(1) 元素 a 和元素 b 是兄弟元素的情况

相关代码如下所示：

```

4833     var al, bl,
4834         ap = [],
4835         bp = [],
4836         aup = a.parentNode,
4837         bup = b.parentNode,
4838         cur = aup;
4839
4840         // If the nodes are siblings (or identical) we can do a quick check
4841         if ( aup === bup ) {
4842             return siblingCheck( a, b );
4843

```

第 4836 ~ 4837 行、第 4841 ~ 4842 行：变量 aup 是元素 a 的父元素，变量 bup 是元素 b 的父元素。如果变量 aup 与变量 bup 相等，说明元素 a 和元素 b 是兄弟元素，则调用函数 siblingCheck() 比较元素 a 和元素 b 的文档位置。函数 siblingCheck(a, b, ret) 负责比较兄弟元素的文档位置，稍后会看到该函数的源码实现和分析。

(2) 没有找到父元素的情况

相关代码如下所示：

```

4844         // If no parents were found then the nodes are disconnected
4845         } else if ( !aup ) {

```

```

4846         return -1;
4847
4848     } else if ( !bup ) {
4849         return 1;
4850     }
4851

```

第 4845 ~ 4850 行：如果元素 a 没有父元素，则认为元素 a 不在文档中，返回 -1，元素 a 将排在元素 b 之前；如果元素 b 没有父元素，则认为元素 b 不在文档中，返回 1，元素 b 将排在元素 a 之前。

(3) 查找元素 a 和元素 b 的祖先元素

相关代码如下所示：

```

4852     //Otherwise they're somewhere else in the tree so we need
4853     //to build up a full list of the parentNodes for comparison
4854     while ( cur ) {
4855         ap.unshift( cur );
4856         cur = cur.parentNode;
4857     }
4858
4859     cur = bup;
4860
4861     while ( cur ) {
4862         bp.unshift( cur );
4863         cur = cur.parentNode;
4864     }
4865

```

(4) 比较祖先元素的文档位置

相关代码如下所示：

```

4866     al = ap.length;
4867     bl = bp.length;
4868
4869     //Start walking down the tree looking for a discrepancy
4870     for ( var i = 0; i < al && i < bl; i++ ) {
4871         if ( ap[i] !== bp[i] ) {
4872             return siblingCheck( ap[i], bp[i] );
4873         }
4874     }
4875

```

第 4866 ~ 4874 行：从最顶层的祖先元素开始向下遍历，如果祖先元素 ap[i] 与 bp[i] 不是同一个元素，那么它们必然是兄弟元素，此时可以通过比较两个祖先元素的文档位置，来确定元素 a 和元素 b 的相对位置。

(5) 元素 a 和元素 b 的文档深度不一致的情况

相关代码如下所示：

```

4876     //We ended someplace up the tree so do a sibling check
4877     return i === al ?
4878         siblingCheck( a, bp[i], -1 ) :

```

```

4879         siblingCheck( ap[i], b, 1 );
4880     };
4881

```

第 4877 ~ 4878 行：如果元素 a 的文档深度较小，此时元素 a 与元素 b 的祖先元素 bp[i] 要么是兄弟元素，要么是同一个元素，可以调用函数 `siblingCheck(a, b, ret)` 比较文档位置。

第 4877 ~ 4879 行：如果元素 b 的文档深度较小，此时元素 a 的祖先元素 ap[i] 与元素 b 要么是兄弟元素，要么是同一个元素，可以调用函数 `siblingCheck(a, b, ret)` 比较文档位置。

(6) `siblingCheck(a, b, ret)`

函数 `siblingCheck(a, b, ret)` 负责比较兄弟元素的文档位置。该函数从元素 a 向后遍历 (`nextSibling`)，如果遇到元素 b，说明元素 a 在元素 b 之前，则返回 -1；如果一直没遇到，说明元素 a 在元素 b 之后，则返回 1。

相关代码如下所示：

```

4882     siblingCheck = function( a, b, ret ) {
4883         if ( a === b ) {
4884             return ret;
4885         }
4886
4887         var cur = a.nextSibling;
4888
4889         while ( cur ) {
4890             if ( cur === b ) {
4891                 return -1;
4892             }
4893
4894             cur = cur.nextSibling;
4895         }
4896
4897         return 1;
4898     };
4899 }

```

3.10.3 Sizzle.contains(a, b)

工具方法 `Sizzle.contains(a, b)` 负责检测元素 a 是否包含元素 b。该方法通过调用原生方法 `contains()` 或 `compareDocumentPosition()` 实现。原生方法 `contains()` 用于检测一个元素是否包含另一个元素；原生方法 `compareDocumentPosition()` 用于比较两个元素的文档位置，更多信息请访问以下网址：

<http://ejohn.org/blog/comparing-document-position/>

http://www.quirksmode.org/dom/w3c_core.html#miscellaneous

相关代码如下所示：

```

5242 if ( document.documentElement.contains ) {
5243     Sizzle.contains = function( a, b ) {
5244         return a !== b && (a.contains ? a.contains(b) : true);
5245     };
5246
5247 } else if ( document.documentElement.compareDocumentPosition ) {

```

```

5248     Sizzle.contains = function( a, b ) {
5249         return !(a.compareDocumentPosition(b) & 16);
5250     };
5251
5252 } else {
5253     Sizzle.contains = function() {
5254         return false;
5255     };
5256 }

```

3.10.4 Sizzle.error(msg)

工具方法 Sizzle.error(msg) 用于抛出一个含有选择器表达式语法错误信息的异常。

相关代码如下所示：

```

4178 Sizzle.error = function( msg ) {
4179     throw new Error( "Syntax error, unrecognized expression: " + msg );
4180 };

```

3.10.5 Sizzle.getText(elem)

工具方法 Sizzle.getText(elem) 用于获取元素集合中所有元素合并后的文本内容。

相关代码如下所示：

```

4182 /**
4183 * Utility function for retrieving the text value of an array of DOM nodes
4184 * @param {Array|Element} elem
4185 */
4186 var getText = Sizzle.getText = function( elem ) {
4187     var i, node,
4188         nodeType = elem.nodeType,
4189         ret = "";
4190
4191     if ( nodeType ) {
4192         if ( nodeType === 1 || nodeType === 9 ) {
4193             //Use textContent || innerText for elements
4194             if ( typeof elem.textContent === 'string' ) {
4195                 return elem.textContent;
4196             } else if ( typeof elem.innerText === 'string' ) {
4197                 //Replace IE's carriage returns
4198                 return elem.innerText.replace( rReturn, '' );
4199             } else {
4200                 //Traverse it's children
4201                 for ( elem = elem.firstChild; elem; elem = elem.nextSibling ) {
4202                     ret += getText( elem );
4203                 }
4204             }
4205         } else if ( nodeType === 3 || nodeType === 4 ) {
4206             return elem.nodeValue;
4207         }
4208     } else {
4209

```

```

4210      // If no nodeType, this is expected to be an array
4211      for ( i = 0; (node = elem[i]); i++ ) {
4212          // Do not traverse comment nodes
4213          if ( node.nodeType !== 8 ) {
4214              ret += getText( node );
4215          }
4216      }
4217  }
4218  return ret;
4219 };

```

第 4191 ~ 4207 行：如果参数 elem 是元素，则尝试读取属性 `textContent` 或 `innerText`，如果不支持则遍历子元素，递归调用工具函数 `getText(elem)` 来获取每个子元素的文本内容，并合并；如果参数 elem 是 Text 节点或 CDATASection 节点，则直接返回节点值 `nodeValue`。

第 4208 ~ 4217 行：否则认为参数 elem 是元素集合，遍历该元素集合，递归调用函数 `getText(elem)` 获取每个元素的文本内容，并合并。

第 4218 行：最后返回合并后的文本内容。

3.11 便捷方法

3.11.1 Sizzle.matches(expr, set)

便捷方法 `Sizzle.matches(expr, set)` 使用指定的选择器表达式 `expr` 对元素集合 `set` 进行过滤，并返回过滤结果。

该方法通过简单地调用函数 `Sizzle(selector, context, results, seed)` 来实现，调用时会将元素集合 `set` 作为参数 `seed` 传入。

相关代码如下所示：

```

4043 Sizzle.matches = function( expr, set ) {
4044     return Sizzle( expr, null, null, set );
4045 };

```

3.11.2 Sizzle.matchesSelector(node, expr)

便捷方法 `Sizzle.matchesSelector(node, expr)` 用于检查某个元素 `node` 是否匹配选择器表达式 `expr`。

如果浏览器支持原生方法 `matchesSelector()`、`mozMatchesSelector()`、`webkitMatchesSelector()`、`msMatchesSelector()` 中的一种，则尝试调用原生方法检查元素与选择器表达式是否匹配；如果浏览器不支持原生方法，或者支持但是检查失败（抛出异常），则调用函数 `Sizzle(selector, context, results, seed)`，检查其返回值的长度是否大于 0，调用时会将元素 `node` 封装成数组作为参数 `seed` 传入。

相关代码如下所示：

```

4047 Sizzle.matchesSelector = function( node, expr ) {
4048     return Sizzle( expr, null, null, [node] ).length > 0;
4049 };

```

```

5095 (function() {
5096     var html = document.documentElement,
5097         matches = html.matchesSelector || html.mozMatchesSelector || html.
5098 webkitMatchesSelector || html.msMatchesSelector;
5099     if ( matches ) {
5100         // Check to see if it's possible to do matchesSelector
5101         // on a disconnected node (IE 9 fails this)
5102         var disconnectedMatch = !matches.call( document.createElement( "div" ),
5103             "div" ),
5104             pseudoWorks = false;
5105         try {
5106             // This should fail with an exception
5107             // Gecko does not error, returns false instead
5108             matches.call( document.documentElement, "[test!='']:sizzle" );
5109         } catch( pseudoError ) {
5110             pseudoWorks = true;
5111         }
5112     }
5113     Sizzle.matchesSelector = function( node, expr ) {
5114         // Make sure that attribute selectors are quoted
5115         expr = expr.replace(/=\s*([\"\\'])\s*/g, ="$1']");
5116
5117         if ( !Sizzle.isXML( node ) ) {
5118             try {
5119                 if ( pseudoWorks || !Expr.match.PSEUDO.test( expr ) &&
5120 !/=/.test( expr ) ) {
5121                     var ret = matches.call( node, expr );
5122
5123                     // IE 9's matchesSelector returns false on
5124                     // As well, disconnected nodes are said to be
5125                     // disconnected nodes
5126                     if ( ret || !disconnectedMatch ||
5127                         // fragment in IE 9, so check for that
5128                         node.document && node.documentElement.nodeType !== 11 ) {
5129                         return ret;
5130                     }
5131                     } catch(e) {}
5132                 }
5133
5134             return Sizzle(expr, null, null, [node]).length > 0;
5135         };
5136     }
5137 })();

```

3.12 jQuery 扩展

3.12.1 暴露 Sizzle 给 jQuery

下面的代码将 Sizzle 的方法和属性暴露给了 jQuery：

```

5288 // EXPOSE
5289 // Override sizzle attribute retrieval
5290 Sizzle.attr = jQuery.attr;
5291 Sizzle.selectors.attrMap = {};
5292 jQuery.find = Sizzle;
5293 jQuery.expr = Sizzle.selectors;
5294 jQuery.expr[":"] = jQuery.expr.filters;
5295 jQuery.unique = Sizzle.uniqueSort;
5296 jQuery.text = Sizzle.getText;
5297 jQuery.isXMLDoc = Sizzle.isXML;
5298 jQuery.contains = Sizzle.contains;

```

3.12.2 .find(selector)

方法`.find(selector)`用于获取匹配元素集合中的每个元素的后代元素，可以用一个选择器表达式、jQuery 对象或 DOM 元素来过滤查找结果，并用匹配的元素构造一个新的 jQuery 对象作为返回值。该方法通过调用函数`Sizzle(selector, context, results, seed)`来实现，并在后者的基础上增加了对多上下文和链式语法的支持。

方法`.find(selector)`执行的 2 个关键步骤如下：

- 1) 如果参数`select`是 jQuery 对象或 DOM 元素，则检查其是否是当前元素集合中某个元素的后代元素，是则保留，不是则丢弃。
- 2) 如果参数`selector`是字符串，则遍历当前元素集合，以每个元素为上下文，调用方法`jQuery.find(selector, context, results, seed)`也就是`Sizzle(selector, context, results, seed)`，查找匹配的后代元素，并将查找结果合并、去重。

相关代码如下所示：

```

5319 jQuery.fn.extend({
5320     find: function( selector ) {
5321         var self = this,
5322             i, l;
5323
5324         if ( typeof selector !== "string" ) {
5325             return jQuery( selector ).filter(function() {
5326                 for ( i = 0, l = self.length; i < l; i++ ) {
5327                     if ( jQuery.contains( self[ i ], this ) ) {
5328                         return true;
5329                     }
5330                 }
5331             });
5332         }
5333
5334         var ret = this.pushStack( "", "find", selector ),
5335             length, n, r;
5336
5337         for ( i = 0, l = this.length; i < l; i++ ) {
5338             length = ret.length;
5339             jQuery.find( selector, this[i], ret );
5340
5341             if ( i > 0 ) {

```

```

5342         // Make sure that the results are unique
5343         for ( n = length; n < ret.length; n++ ) {
5344             for ( r = 0; r < length; r++ ) {
5345                 if ( ret[r] === ret[n] ) {
5346                     ret.splice(n--, 1);
5347                     break;
5348                 }
5349             }
5350         }
5351     }
5352 }
5353
5354     return ret;
5355 },
5470 });

```

第 5320 行：定义方法 `.find(selector)`，参数 `selector` 可以是选择器表达式，也可以是 jQuery 对象或 DOM 元素。

第 5324 ~ 5332 行：如果参数 `selector` 不是字符串，则认为该参数是 jQuery 对象或 DOM 元素，此时先将该参数统一封装为一个新 jQuery 对象，然后遍历新 jQuery 对象，检查其中的元素是否是当前 jQuery 对象中某个元素的后代元素，如果是则保留，不是则丢弃。最后返回含有新 jQuery 对象子集的另一个新 jQuery 对象。

第 5334 行：调用方法 `.pushStack(elements, name, arguments)` 构造一个新的空 jQuery 对象，并将其作为返回值，后面找到的元素都将被添加到该 jQuery 对象中。

第 5337 ~ 5352 行：如果参数 `selector` 是字符串，则遍历当前元素集合，以每个元素为上下文，调用方法 `jQuery.find(selector, context, results, seed)` 也就是 `Sizzle(selector, context, results, seed)`，查找匹配的后代元素，并将查找结果合并、去重。

第 5341 ~ 5351 行：从当前元素集合的第 2 个元素开始，遍历新找到的元素数组，移除其中与已找到的元素相等的元素，以确保找到的元素是唯一的。变量 `length` 表示已找到的元素集合的长度，也就是新找到的元素被插入的开始位置；移除时通过将循环变量 `n` 自减 1，确保接下来的遍历不会漏掉元素。

3.12.3 .has(target)

方法 `.has(target)` 用当前 jQuery 对象的子集构造一个新 jQuery 对象，其中只保留子元素可以匹配参数 `target` 的元素。参数 `target` 可以是选择器表达式、jQuery 对象或 DOM 元素。该方法通过调用方法 `.filter(selector)` 遍历当前匹配元素集合，通过调用 `jQuery.contains(a, b)`，也就是 `Sizzle.contains(a, b)`，检查匹配元素是否包含了可以匹配参数 `target` 的子元素。

相关代码如下所示：

```

5319 jQuery.fn.extend({
5357     has: function( target ) {
5358         var targets = jQuery( target );
5359         return this.filter(function() {

```

```

5360         for ( var i = 0, l = targets.length; i < l; i++ ) {
5361             if ( jQuery.contains( this, targets[i] ) ) {
5362                 return true;
5363             }
5364         }
5365     });
5366 },
5470 });

```

第 5357 行：定义方法 `.has(target)`，参数 `target` 可以是选择器表达式、jQuery 对象及元素。

第 5358 行：构造匹配参数 `target` 的 jQuery 对象。

第 5359 ~ 5365 行：调用方法 `.filter(selector)` 遍历当前匹配元素集合，检查每个匹配元素是否包含了参数 `target` 所匹配的某个元素，如果包含则保留，不包含则丢弃。

3.12.4 `.not(selector)`、`.filter(selector)`

方法 `.not(selector)` 用当前 jQuery 对象的子集构造一个新 jQuery 对象，其中只保留与参数 `selector` 不匹配的元素。参数 `selector` 可以是选择器表达式、jQuery 对象、函数、DOM 元素或 DOM 元素数组。

方法 `.filter(selector)` 用当前 jQuery 对象的子集构造一个新 jQuery 对象，其中只保留与参数 `selector` 匹配的元素。参数 `selector` 可以是选择器表达式、jQuery 对象、函数、DOM 元素或 DOM 元素数组。

方法 `.not(selector)` 和 `.filter(selector)` 通过调用函数 `winnow(elements, qualifier, keep)` 对当前匹配元素集合进行过滤，并用其返回值构造一个新 jQuery 对象。不过，这两个方法的过滤行为正好相反，这种差异通过调用函数 `winnow(elements, qualifier, keep)` 时传入不同的参数 `keep` 来实现。

相关代码如下所示：

```

5319 jQuery.fn.extend({
5368     not: function( selector ) {
5369         return this.pushStack( winnow(this, selector, false), "not",
5370     ),
5371     },
5372     filter: function( selector ) {
5373         return this.pushStack( winnow(this, selector, true), "filter",
5374     ),
5470 });
5590 // Implement the identical functionality for filter and not
5591 function winnow( elements, qualifier, keep ) {
5592
5593     // Can't pass null or undefined to indexOf in Firefox 4
5594     // Set to 0 to skip string check

```

```

5595     qualifier = qualifier || 0;
5596
5597     if ( jQuery.isFunction( qualifier ) ) {
5598         return jQuery.grep(elements, function( elem, i ) {
5599             var retVal = !qualifier.call( elem, i, elem );
5600             return retVal === keep;
5601         });
5602
5603     } else if ( qualifier.nodeType ) {
5604         return jQuery.grep(elements, function( elem, i ) {
5605             return ( elem === qualifier ) === keep;
5606         });
5607
5608     } else if ( typeof qualifier === "string" ) {
5609         var filtered = jQuery.grep(elements, function( elem ) {
5610             return elem.nodeType === 1;
5611         });
5612
5613         if ( isSimple.test( qualifier ) ) {
5614             return jQuery.filter(qualifier, filtered, !keep);
5615         } else {
5616             qualifier = jQuery.filter( qualifier, filtered );
5617         }
5618     }
5619
5620     return jQuery.grep(elements, function( elem, i ) {
5621         return ( jQuery.inArray( elem, qualifier ) >= 0 ) === keep;
5622     });
5623 }

```

第 5591 行：函数 `winnow(elements, qualifier, keep)` 负责过滤元素集合，它接受 3 个参数：

□ 参数 `elements`: 待过滤的元素集合。

□ 参数 `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。

□ 参数 `keep`: 布尔值。如果为 `true`，则保留匹配元素，如果为 `false`，则保留不匹配元素。

第 5597 ~ 5601 行：如果参数 `qualifier` 是函数，则调用方法 `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 遍历元素集合 `elements`，在每个元素上执行该函数，然后将返回值与参数 `keep` 进行比较，如果一致则保留，不一致则丢弃。

第 5603 ~ 5606 行：如果参数 `qualifier` 是 DOM 元素，则调用方法 `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 遍历元素集合 `elements`，检查其中的每个元素是否与参数 `qualifier` 相等，然后将检查结果与参数 `keep` 进行比较，如果一致则保留，不一致则丢弃。

第 5608 ~ 5622 行：如果参数 `qualifier` 是字符串，则先过滤出与该参数匹配的元素集合，然后调用方法 `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 遍历元素集合 `elements`，检查其中的每个元素是否在过滤结果中，然后将检查结果与参数 `keep` 进行比较，如果一致则保留，不一致则丢弃。

第 5620 ~ 5622 行：如果参数 `qualifier` 是 DOM 元素数组或 jQuery 对象，则调用方法

`jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 遍历元素集合 `elements`, 检查其中的每个元素是否在参数 `qualifier` 中, 然后将检查结果与参数 `keep` 进行比较, 如果一致则保留, 不一致则丢弃。

关于方法 `jQuery.grep(array, function(elementOfArray, indexInArray)[, invert])` 的介绍和分析请参见 2.8.8 节。

3.12.5 .is(selector)

方法 `.is(selector)` 用选择器表达式、DOM 元素、jQuery 对象或函数来检查当前匹配元素集合, 只要其中某个元素可以匹配给定的参数就返回 `true`。

相关代码如下所示:

```
5293  jQuery.expr = Sizzle.selectors;
5310      POS = jQuery.expr.match.POS,
5319  jQuery.fn.extend({
5376      is: function( selector ) {
5377          return !selector && (
5378              typeof selector === "string" ?
5379                  // If this is a positional selector, check membership in the
5380                  // returned set
5381                  // so $("p:first").is("p:last") won't return true for a doc
5382                  // with two "p".
5383                  POS.test( selector ) ?
5384                      jQuery( selector, this.context ).index( this[0] ) >= 0 :
5385                      jQuery.filter( selector, this ).length > 0 :
5386                          this.filter( selector ).length > 0 );
5385      },
5470  });

华章白节
```

第 5376 行: 定义方法 `.is(selector)`, 其中参数 `selector` 可以是选择器表达式、DOM 元素、jQuery 对象或函数。

第 5378 行、第 5381 行、第 5382 行: 如果参数 `selector` 是字符串, 并且含有位置伪类, 则先调用构造函数 `jQuery(selector[, context])` 查找参数 `selector` 匹配的元素集合, 然后检查当前匹配元素集合中的第一个元素是否在查找结果中, 如果在结果中则返回 `true`, 如果不在则返回 `false`。

第 5378 行、第 5381 行、第 5383 行: 如果参数 `selector` 是字符串, 并且不含有位置伪类, 则调用方法 `jQuery.filter(expr, elems, not)` 用参数 `selector` 过滤当前匹配元素集合, 然后检查过滤结果中是否仍有元素, 如果有则返回 `true`, 如果没有则返回 `false`。

第 5378 行、第 5384 行: 如果参数 `selector` 不是字符串, 则可能是 DOM 元素、jQuery 对象或函数, 先调用方法 `.filter(selector)` 用参数 `selector` 过滤当前匹配元素集合, 然后检查过滤结果中是否仍有元素, 如果有则返回 `true`, 如果没有则返回 `false`。

3.12.6 .closest(selectors, context)

方法 `.closest(selectors [, context])` 用于在当前匹配元素集合和它们的祖先元素中查找与参数 `selectors` 匹配的最近元素，并用查找结果构造一个新 `jQuery` 对象。

方法 `.closest(selectors [, context])` 与 `.parents([selector])` 的行为相似，都是沿着 DOM 树向上查找匹配元素，需要注意的是两者的差别，具体如表 3-16 所示。

表 3-16 `.closest(selectors [, context])`、`.parents([selector])`

序号	<code>.closest(selectors [, context])</code>	<code>.parents([selector])</code>
1	从每个匹配元素开始	从每个匹配元素的父元素开始
2	沿着 DOM 树向上遍历，直到找到匹配参数 <code>selectors</code> 的元素	沿着 DOM 树向上遍历，查找所有与参数 <code>selectors</code> 匹配的元素

相关代码如下所示：

```

5293  jQuery.expr = Sizzle.selectors;
5310      POS = jQuery.expr.match.POS,
5319  jQuery.fn.extend({
5387      closest: function( selectors, context ) {
5388          var ret = [], i, l, cur = this[0];
5389
5390          //Array (deprecated as of jQuery 1.7)
5391          if ( jQuery.isArray( selectors ) ) {
5392              var level = 1;
5393
5394              while ( cur && cur.ownerDocument && cur !== context ) {
5395                  for ( i = 0; i < selectors.length; i++ ) {
5396
5397                      if ( jQuery( cur ).is( selectors[ i ] ) ) {
5398                          ret.push({ selector: selectors[ i ], elem: cur, level: level });
5399                      }
5400                  }
5401
5402                  cur = cur.parentNode;
5403                  level++;
5404              }
5405
5406              return ret;
5407          }
5408
5409          // String
5410          var pos = POS.test( selectors ) || typeof selectors !== "string" ?
5411              jQuery( selectors, context || this.context ) :
5412              0;
5413
5414          for ( i = 0, l = this.length; i < l; i++ ) {
5415              cur = this[i];
5416
5417              while ( cur ) {

```

```

5418             if ( pos ? pos.index(cur) > -1 : jQuery.find.matchesSelector
5419                 (cur, selectors) ) {
5420                     ret.push( cur );
5421                     break;
5422             } else {
5423                 cur = cur.parentNode;
5424                 if ( !cur || !cur.ownerDocument || cur === context ||
5425                     cur.nodeType === 11 ) {
5426                     break;
5427                 }
5428             }
5429         }
5430     ret = ret.length > 1 ? jQuery.unique( ret ) : ret;
5431
5432     return this.pushStack( ret, "closest", selectors );
5433 },
5434
5470 });

```

第 5387 行：定义方法 .closest(selectors [, context])，它接受 2 个参数：

- 参数 **selectors**：用于匹配 DOM 元素，可选值有：选择器表达式、jQuery 对象、DOM 元素、数组。
- 参数 **context**：可选的上下文，用于限定查找范围。

第 5391 ~ 5407 行：如果参数 **selectors** 是数组，则从当前匹配元素集合中的第一个元素开始沿着 DOM 树向上遍历，查找与数组 **selector** 中的元素匹配的元素，直到遇到上下文 **context** 或 **document** 对象为止。如果找到与数组 **selector** 中的元素匹配的元素，则放入数组 **ret** 中，其格式为：

```
{ selector: 参数 selector 中的元素, elem: 匹配元素, level: 向上查找的层级 }
```

第 5414 ~ 5429 行：遍历当前元素集合，在每个匹配元素和它的祖先元素中查找与参数 **selectors** 匹配的最近元素。

第 5417 ~ 5428 行：在向上遍历的过程中，如果找到了与参数 **selectors** 匹配的元素，则把它插入数组 **ret**，然后跳出 **while** 循环，继续在下一个匹配元素和它的祖先元素中查找。

第 5422 ~ 5427 行：如果当前元素 **cur** 不匹配参数 **selectors**，则读取它的父元素，继续向上遍历。如果父元素不存在，或者父元素不在文档中，或者父元素是上下文，或者父元素是文档片段，则跳出 **while** 循环，继续在下一个匹配元素和它的祖先元素中查找。

第 5431 行：如果找到了多个匹配参数 **selectors** 的元素，则调用方法 **jQuery.unique(results)**，也就是 **Sizzle.uniqueSort(results)**，执行排序和去重操作。

第 5433 行：最后用找到的元素构造一个新 **jQuery** 对象，并返回。

3.12.7 .index(elem)

方法 **.index(elem)** 用于判断元素在元素集合中的下标位置。该方法的行为随参数 **elem**

的不同而不同，如表 3-17 所示。

表 3-17 .index(elem)

序号	参数 elem	行为
1	没有传入	返回第一个匹配元素相对于其兄弟元素的下标位置
2	选择器表达式	返回第一个匹配元素在选择器表达式所匹配的元素集合中的位置
3	DOM 元素	返回 DOM 元素在当前匹配元素集合中的位置
4	jQuery 对象	返回 jQuery 对象的第一个元素在当前匹配元素集合中的位置

相关代码如下所示：

```

5319  jQuery.fn.extend({
5436      // Determine the position of an element within
5437      // the matched set of elements
5438      index: function( elem ) {
5439
5440          // No argument, return index in parent
5441          if ( !elem ) {
5442              return ( this[0] && this[0].parentNode ) ? this.prevAll().length : -1;
5443          }
5444
5445          // index in selector
5446          if ( typeof elem === "string" ) {
5447              return jQuery.inArray( this[0], jQuery( elem ) );
5448          }
5449
5450          // Locate the position of the desired element
5451          return jQuery.inArray(
5452              // If it receives a jQuery object, the first element is used
5453              elem.jquery ? elem[0] : elem, this );
5454      },
5470  });

```

第 5441 ~ 5443 行：如果没有参数，则返回当前匹配元素集合中第一个元素相对于其兄弟元素的位置。如果第一个元素没有父元素，则返回 -1。

第 5446 ~ 5448 行：如果参数 elem 是字符串，则调用构造函数 `jQuery(selector[, context])` 查找参数 elem 匹配的元素集合，然后调用方法 `jQuery.inArray(elem, array, i)` 获取当前匹配元素集合中第一个元素在查找结果中的位置，并返回。

第 5451 ~ 5453 行：如果参数 elem 是 DOM 元素，则调用方法 `jQuery.inArray(elem, array, i)` 获取参数 elem 在当前元素匹配集合中的位置，并返回；如果参数 elem 是 jQuery 对象，则调用方法 `jQuery.inArray(elem, array, i)` 获取参数 elem 的第一个元素在当前匹配元素集合中的位置，并返回。

3.12.8 .add(selector, context)

方法 `.add(selector, context)` 用当前 jQuery 对象中的元素和传入的参数构造一个新 jQuery

对象。构造函数 `jQuery()` 可以接受的参数格式，该方法都可以接受。另外，该方法不会改变当前 `jQuery` 对象。

相关代码如下所示：

```

5319 jQuery.fn.extend({
5456     add: function( selector, context ) {
5457         var set = typeof selector === "string" ?
5458             jQuery( selector, context ) :
5459             jQuery.makeArray( selector && selector.nodeType ?
[ selector ] : selector ),
5460             all = jQuery.merge( this.get(), set );
5461
5462         return this.pushStack( isDisconnected( set[0] ) || isDisconnected
( all[0] ) ?
5463             all :
5464             jQuery.unique( all ) );
5465     },
5470 });
5472 // A painfully simple check to see if an element is disconnected
5473 // from a document (should be improved, where feasible).
5474 function isDisconnected( node ) {
5475     return !node || !node.parentNode || node.parentNode.nodeType === 11;
5476 }

```

第 5457 ~ 5459 行：如果参数 `selector` 是字符串，则调用构造函数 `jQuery(selector[, context])` 查找与之匹配的元素集合，并赋值给变量 `set`；否则调用方法 `jQuery.makeArray(array, results)` 将参数 `selector` 转换为数组。

第 5460 行：调用方法 `jQuery.merge(first, second)` 将当前 `jQuery` 对象中的元素和元素集合 `set` 的元素合并，并赋值给变量 `all`。

第 5462 ~ 5464 行：用合并后的元素集合 `all` 构造一个新 `jQuery` 对象。如果元素集合 `set` 的第一个元素或合并后的元素集合 `all` 中第一个元素不在文档中，则可以直接用元素集合 `all` 构造新 `jQuery` 对象；否则，需要先调用方法 `jQuery.unique(results)`，也就是 `Sizzle.uniqueSort(results)`，对合并后的元素集合 `all` 进行排序和去重，然后再构造 `jQuery` 对象。

第 5474 ~ 5476 行：函数 `isDisconnected(node)` 用于简单地判断一个元素是否不在文档中。如果元素不存在，或者父元素不存在，或者父元素是文档片段，则返回 `true`。

3.12.9 `jQuery.filter(expr, elems, not)`

方法 `jQuery.filter(expr, elems, not)` 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。

相关代码如下所示：

```

5540 jQuery.extend({
5541     filter: function( expr, elems, not ) {
5542         if ( not ) {

```

```

5543         expr = ":not(" + expr + ")";
5544     }
5545
5546     return elems.length === 1 ?
5547         jQuery.find.matchesSelector(elems[0], expr) ? [ elems[0] ] : [] :
5548         jQuery.find.matches(expr, elems);
5549     },
5588 });

```

第 5541 行：定义方法 `jQuery.filter(expr, elems, not)`，它接受 3 个参数：

- 参数 `expr`: 选择器表达式。
- 参数 `elems`: 待过滤的元素集合。
- 参数 `not`: 布尔值。如果是 `true`，则保留不匹配元素，否则默认保留匹配元素。

第 5542 ~ 5544 行：修正选择器表达式 `expr`。

第 5546、5547 行：如果元素集合 `elems` 中只有一个元素，则调用方法 `jQuery.find.matchesSelector(node, expr)`，也就是 `Sizzle.matchesSelector(node, expr)`，来检查元素是否匹配选择器表达式 `expr`。

第 5546 ~ 5548 行：如果含有多个元素，则调用方法 `jQuery.find.matches(expr, set)`，也就是 `Sizzle.matches(expr, set)`，来用选择器表达式 `expr` 对元素集合 `elems` 进行过滤。

3.12.10 :animated

jQuery 的动画模块扩展了伪类 `:animated`，用于检测 DOM 元素是否正在执行动画。在对应的伪类过滤函数 `jQuery.expr.filters.animated` 中，会遍历全局动画函数数组 `jQuery.timers`，检查每个动画函数的属性 `elem` 是否是当前元素；如果某个动画函数的属性 `elem` 是当前元素，则表示当前元素正在执行动画。更多信息请参考第 14 章。

相关代码如下所示：

```

8842 if ( jQuery.expr && jQuery.expr.filters ) {
8843     jQuery.expr.filters.animated = function( elem ) {
8844         return jQuery.grep(jQuery.timers, function( fn ) {
8845             return elem === fn.elem;
8846         }).length;
8847     };
8848 }

```

3.12.11 hidden、:visible

jQuery 的样式操作模块扩展了伪类 `:hidden` 和 `:visible`，用来判断 DOM 元素是否占据布局空间。在对应的伪类过滤函数 `jQuery.expr.filters.hidden/visible` 中，通过判断 DOM 元素的可见宽度 `offsetWidth` 和可见高度 `offsetHeight` 是否为 0，来判断该元素是否占据布局空间。

注意：设置样式 `visibility` 为 `hidden`，或设置样式 `opacity` 为 0 后，DOM 元素仍然会占据布局空间。

相关代码如下所示：

```

6816 if ( jQuery.expr && jQuery.expr.filters ) {
6817     jQuery.expr.filters.hidden = function( elem ) {
6818         var width = elem.offsetWidth,
6819             height = elem.offsetHeight;
6820
6821         return ( width === 0 && height === 0 )
6822             || (!jQuery.support.reliableOffsets && ((elem.style &&
6823             elem.style.display) || jQuery.css( elem, "display" ) === "none"));
6824     };
6825
6826     jQuery.expr.filters.visible = function( elem ) {
6827         return !jQuery.expr.filters.hidden( elem );
6828     };
6829 }

```

第 6817 ~ 6822 行：如果 DOM 元素的可见宽度 offsetWidth 和可见高度 offsetHeight 是 0，则认为该元素不占据布局空间。如果元素的可见高度 offsetHeight 不可靠，则检查样式 display；如果内联样式 display 是 "none"，则认为不占据布局空间；如果未设置行内样式 display，但是计算样式 display 是 "none"，也认为不占据布局空间。

第 6824 ~ 6826 行：通过对伪类过滤函数 jQuery.expr.filters.hidden(elem) 的结果取反，来确定元素是否占据布局空间。

3.13 总结

在本章中，对选择器引擎 Sizzle 做了完整的介绍和分析，总体源码结构见代码清单 3-1，方法功能和调用关系见图 3-1。在本章的最后还介绍和分析了 jQuery 对 Sizzle 的整合和扩展。

选择器表达式由块表达式和块间关系符组成。块表达式分为 3 种：简单表达式、属性表达式、伪类表达式；块间关系符分为 4 种：">" 父子关系、"" 祖先后代关系、"+"> 紧挨着的兄弟元素、" ~ " 之后的所有兄弟元素；块表达式和块间关系符组成了层级表达式。见图 3-3。

Sizzle(selector, context, results, seed) 用于查找与选择器表达式 selector 匹配的元素集合。如果浏览器支持原生方法 querySelectorAll()，则调用该方法查找元素，如果不支持，则模拟该方法的行为。执行过程见图 3-2。

正则 **chunker** 用于从选择器表达式中提取块表达式和块间关系符，其分解图和测试用例见图 3-4。

Sizzle.find(expr, context, isXML) 负责查找与块表达式匹配的元素集合。该方法按照表达式类型数组 Sizzle.selectors.order 规定的查找顺序（ID、CLASS、NAME、TAG）逐个尝试查找，如果未找到，则查找上下文的所有后代元素（*）。执行过程见图 3-5。

Sizzle.filter(expr, set, inplace, not) 负责用块表达式过滤元素集合。该方法通过调用过滤函数集 Sizzle.selectors.filter 中的过滤函数来执行过滤操作。执行过程见图 3-6。

Sizzle.selectors 包含了 Sizzle 在查找和过滤过程中用到的正则、查找函数、过滤函数，见图 3-1。

- ❑ **Sizzle.selectors.order** 中定义了查找单个块表达式时的查找顺序，依次是 ID、CLASS、NAME、TAG。其中，CLASS 需要浏览器支持方法 `getElementsByClassName()`。
- ❑ **Sizzle.selectors.match/leftMatch** 中存放了表达式类型和正则的映射，正则用于确定块表达式的类型，并解析其中的参数。解析图见图 3-7 ~ 图 3-14，测试用例见表 3-3 ~ 表 3-11。
- ❑ **Sizzle.selectors.find** 中定义了 ID、CLASS、NAME、TAG 所对应的查找函数。其中，CLASS 需要浏览器支持方法 `getElementsByClassName()`。查找函数会返回数组或 `undefined`，内部通过调用相应的原生方法来查找元素，见表 3-11。
- ❑ **Sizzle.selectors.relative** 中存放了块间关系符和对应的块间关系过滤函数。块间关系过滤函数用于检查映射集 `checkSet` 中的元素是否匹配块间关系符左侧的块表达式。支持的块间关系符和对应的过滤方式见表 3-2。
- ❑ **Sizzle.selectors.preFilter** 中定义了 CLASS、ID、TAG、CHILD、ATTR、PSEUDO、POS 所对应的预过滤函数。预过滤函用于在过滤函数之前修正与过滤操作相关的参数，每种类型的预过滤函数的修正行为见表 3-12；预过滤函数有 3 种返回值，见表 3-13。
- ❑ **Sizzle.selectors.filters** 中定义了一组伪类和对应的伪类过滤函数。伪类过滤函数负责检查元素是否匹配伪类，返回一个布尔值。
- ❑ **Sizzle.selectors.setFilters** 中定义了一组位置伪类和对应的伪类过滤函数。位置伪类过滤函数通过比较下标来确定元素在集合中的位置，返回一个布尔值。
- ❑ **Sizzle.selectors.filter** 中定义了 PSEUDO、CHILD、ID、TAG、CLASS、ATTR、POS 对应的过滤函数。过滤函数负责检查元素是否匹配过滤表达式，返回一个布尔值。