2015年10月14日

Cherry：元素选择器引擎

-CSS3 选择器引擎的JavaScript实现

一、引文

1.1 写作背景

John Resig，做前端的同学肯定再熟悉不过了，jQuery的作者，他在2008年写过一本书《精通JavaScript》，然而这篇文章不是说这本书，也不是说大神John Resig。而是他在《精通JavaScript》中提到过的另一个JavaScript高手Dean Edwards写过的一个CSS选择器引擎：cssQuery。

在Dean Edwards的个人网站http://dean.edwards.name/my/cssQuery/中，作者是这样描述cssQuery的：cssQuery() is a powerful cross-browser JavaScript function that enables querying of a DOM document using CSS selectors. All CSS1 and CSS2 selectors are allowed plus quite a few CSS3 selectors.

cssQuery 是一个功能强大跨浏览器的通过CSS选择器来选择DOM元素的JavaScript库。其实现了全部的CSS1和CSS2的选择器，同时还包括了部分CSS3的选择器。根据代码的完成时间，他完成最终版本是在2005年，这样的CSS选择器引擎在当时已经算是很强大了，要知道虽然CSS2规范在1998年就已经被发布，但是十年后，2008年也没有浏览器完全实现其所有新的功能，而CSS3选择器规范在2011年才被W3C正式定稿，可见cssQuery在当时确实是一个超前、实用的选择器引擎了。就连John Resig在2006年发布的第一个jQuery版本中也能够找到cssQuery的影子。

本文将根据我对cssQuery源代码的阅读，以及对CSS选择器的理解，以及参考jQuery的选择器引擎sizzle（http://sizzlejs.com/），实现自己的CSS选择器引擎Cherry。

1.2 JavaScript考核内容及考核目标

JavaScript的考核内容：JavaScript的考核内容主要是JavaScript权威指南这本书的第一部分，包括JavaScript的：语法结构、类型、值和变量、表达式和运算符、语句、对象、数组、函数、正则表达式。

JavaScript的考核标准：由上面的考核内容，我当时的考虑是应该写一个pure JavaScript的简单JavaScript库，而在短时间要实现一个JavaScript库，再加上个人能力有限，几乎是不可能的，因此选择器几乎每个JavaScript库都有的CSS选择器引擎来做，而且CSS选择器不包含DOM事件（非考核内容）这些非考核内容，但是又几乎涵盖了上面全部考核内容。在和导师商量过后，就把独立完成一个简单且兼容性良好的CSS选择器引擎作为了本阶段的考核标准。

从九月中旬到今天，Cherry选择器引擎也终于写完，代码地址：

https://github.com/Jocs/cherry

二、CSS3选择器及Cherry的实现原理

本章将分为三个部分，第一部分总结W3C中CSS3 选择器的语法、句法和分类等。因为了解CSS3选择器是实现CSS选择器引擎的基础。第二部分论述Cherry相对于cssQuery的优化的地方，及对CSS3选择器的几乎全部支持。最后一个部分分析Cherry的实现原理。

2.1 W3C中关于CSS3选择器的描述总结

2.1.1 选择器的定义

定义：Selectors are patterns that match against elements in a tree, and as such form one of several technologies that can be used to select nodes in an XML document. Selectors have been optimized for use with HTML and XML, and are designed to be usable in performance-critical code.

简单来说选择器就是一种匹配模式，用来匹配DOM树中的特定元素，CSS通过选择器来把样式绑定到特定的DOM元素上，使得元素表现出与之相应的样式特性。

2.1.2 CSS2中选择器与CSS3中选择器的主要区别

1）the list of basic definitions (selector, group of selectors, simple selector, etc.) has been changed; in particular, what was referred to in CSS2 as a simple selector is now called a sequence of simple selectors, and the term "simple selector" is now used for the components of this sequence

CSS3对selector、group of selectors、simple selector等基本名词的定义发生了改变，以前CSS2中的simple selector 现在在CSS3中称作 sequence of simple selectors，而在CSS3中simple selector成为了sequence of simple selector 的基本组成成分。也就是说一个sequence是由多个simple selector 组成，具体这些名词的定义在下面会提到。

2）an optional namespace component is now allowed in element type selectors, the universal selector and attribute selectors

可以为element type selector、universal selector 和 attribute selectors等simple selector 增加一个可选的命名空间。

3）a new combinator has been introduced

combinator 这个专有名词下面也会作解释，在CSS3中引入了一个新的combinator ，叫做General sibling combinator，这个新的combinator通过『~』表示，General sibling combinator 把连个sequences of simple selector 分开，这两个sequences of simple selector 拥有一个相同的父节点，在『~』前面的sequences of simple selector 在DOM树中的位置先于「~」后面的元素。

4）CSS3中增加了三种新的attribute selector和一些新的pseudo-classes。

5）CSS3中引入了pseudo-element的概念。

2.1.3 选择器的语法

选择器是一种语法结构，一个特定的选择器能够一一对应DOM 树中特定的DOM元素集合。选择器的语法结构如下：（包含39中选择器语法）

Pattern	Meaning	Described in section	First defined in CSS level
*	any element	**Universal selector**	2
E	an element of type E	Type selector	1
E[foo='far']	元素E带有一个foo属性，属性值为far	Attribute selectors	2
E[foo]	元素E带有一个foo属性	Attribute selector	2
E[FOO~='bar']	元素E中FOO属性值中包含单词bar	Attribute selector	2
E[foo^="bar"]	an E element whose "foo" attribute value begins exactly with the string "bar"	Attribute selectors	3
E[foo$="bar"]	an E element whose "foo" attribute value ends exactly with the string "bar"	Attribute selectors	3
E[foo*="bar"]	an E element whose "foo" attribute value contains the substring "bar"	Attribute selectors	3

Pattern	Meaning	Described in section	First defined in CSS level
E[fool="en"]	an E element whose "foo" attribute has a hyphen-separated list of values beginning (from the left) with "en"	Attribute selectors	2
E:root	an E element, root of the document	Structural pseudo-classes	3
E:nth-child(n)	an E element, the n-th child of its parent	Structural pseudo-classes	3
E:nth-last-child(n)	an E element, the n-th child of its parent, counting from the last one	Structural pseudo-classes	3
E:nth-of-type(n)	an E element, the n-th sibling of its type	Structural pseudo-classes	3
E:nth-last-of-type(n)	an E element, the n-th sibling of its type, counting from the last one	Structural pseudo-classes	3
E:first-child	an E element, first child of its parent	Structural pseudo-classes	2
E:last-child	an E element, last child of its parent	Structural pseudo-classes	3
E:first-of-type	an E element, first sibling of its type	Structural pseudo-classes	3
E:last-of-type	an E element, last sibling of its type	Structural pseudo-classes	3
E:only-of-type	an E element, only sibling of its type	Structural pseudo-classes	3
E:empty	an E element that has no children (including text nodes)	Structural pseudo-classes	3
E:link **E:visited**	an E element being the source anchor of a hyperlink of which the target is not yet visited (:link) or already visited (:visited)	The link pseudo-classes	1
E:active **E:hover** **E:focus**	an E element during certain user actions	The user action pseudo-classes	1 and 2

Pattern	Meaning	Described in section	First defined in CSS level
E:target	an E element being the target of the referring URI	The target pseudo-class	3
E:lang(fr)	an element of type E in language "fr" (the document language specifies how language is determined)	The :lang() pseudo-class	2
E:enabled **E:disabled**	a user interface element E which is enabled or disabled	The UI element states pseudo-classes	3
E:checked	a user interface element E which is checked (for instance a radio-button or checkbox)	The UI element states pseudo-classes	3
E::first-line	the first formatted line of an E element	The ::first-line pseudo-element	1
E::first-letter	the first formatted letter of an E element	The ::first-letter pseudo-element	1
E::before	generated content before an E element	The ::before pseudo-element	2
E::after	generated content after an E element	The ::after pseudo-element	2
E.warning	an E element whose class is "warning" (the document language specifies how class is determined).	Class selectors	1
E#myid	an E element with ID equal to "myid".	ID selectors	1
E ~ F	an F element preceded by an E element	General sibling combinator	3
E:not(s)	an E element that does not match simple selector s	Negation pseudo-class	3
E F	an F element descendant of an E element	Descendant combinator	1
E > F	an F element child of an E element	Child combinator	2

Pattern	Meaning	Described in section	First defined in CSS level
E + F	an F element immediately preceded by an E element	Adjacent sibling combinator	2
E:only-child	an E element, only child of its parent	Structural pseudo-classes	3

2.1.4 选择器的句法

一个选择器是由一个或多个sequences of simple selector组成的选择器链，而sequences of simple selector是通过combinators来分开，（combinator包括「' '」「'>'」「'+'」「'~'」）。而一个pseudo-element可以加在选择器中任意一个simple selector后面。

一个sequence of simple selectors 是由simple selector（简单选择器）组成的链。一个sequence of simple selector通常以一个element type selector或者Universal selector开头。而且通常一个sequence of simple selectors 也只能够包含一个element type selector 或者universal selector。

simple selector 是 type selector, universal selector, attribute selector, class selector, ID selector, or pseudo-class. 中的任意一种。也就是在选择器语法的表格中，（上表），除了combinator其他都是simple selector 。

如下选择器：body > *.red + *#selected (当然实际中没必要这么写，直接#selected就搞定了)

上面的选择器包含两个combinators，「 > 」「 + 」，把选择器分成了三个sequence of simple selector: (body、*.red、*#selected)。每个sequence of simple selector 都是由element type selector 或者 Universal Selector开头的，而且每个sequence of simple selector 中也只包含一个Element type selector 或者 Universal Selector，比如body中只包含body这个Element Type Selector 而 *.red中只包含*这个通配符。通配符如果不是sequence of

simple 中唯一的simple selector ，通常可以省略的。上面的选择器也就是可以简写成如下：

body > .red + #selected

但是通常为了避免不必要的混淆，建议保留通配符，在选择器引擎中，在对选择器解析前也都是要把省略的「*」先加上去。

2.1.5 Groups of selectors

把多个选择器通过「逗号」组合在一起就组成了一个Groups of selectors，Groups of selectors通常用于样式表对相同样式的元素简写中使用。

比如：

h1 { font-family: sans-serif }

h2 { font-family: sans-serif }

h3 { font-family: sans-serif }

和 以下简写形式效果一样：

h1, h2, h3 { font-family: sans-serif }

2.1.6 Combinators

在W3C关于CSS选择器的最新文档中，现已有四个Combinatros，其中三个是CSS2添加，一个是CSS3添加的。

1）Descendant combinator

在CSS2中引入，Descendant combinator 通常被称作后代选择器，因为其用来描述父元素的所有子孙元素，一个Descendant combinator 是一个空白符（whitesapce）用来把 sequences of simple selectors 分开，例如：

body *.red (一个小技巧，如果为了格式上的好看，通常空白符可以多个，因为在选择器引擎中会处理掉多余的空白符而只留下一个）

2）Child combinators

在CSS2中引入，顾名思义，Child combinators 就是子选择器，用来选择父元素下面的子元素，通常用一个大于号表示「>」。它的作用除了其用来选择子元素外，还具有和其他combinators一样的功能，就是作为sequences of simple selectors的分隔符。

3） Sibling combinators

在CSS2中引入，在CSS3中有增加了一个新的Sibling combinators，现目前一共包含两个Sibling combinators。Adjacent sibling combinator 和 General sibling combinator。

Adjacent sibling combinator

这个combinator在CSS2中引入，通过加号表示「+」，其作用是用来选择相邻的下一个兄弟元素，如：

#selected + p

General sibling combinator

这个选择器在上面已经提到过，通过波浪符号表示「~」，其是CSS3新引入的combinator，其作用是用来选择某元素后面的所有指定的兄弟元素。波浪符号「~」前后的紧跟的simple selector具有相同的父元素。只是波浪符号后面的元素在DOM树中的位置排在了波浪符号前面的元素后面。

例如：

#selected ~ p

其含义是用来选择ID值为selected的元素后面的所有 tagName为「P」的兄弟元素。

2.2 Cherry 对 cssQuery改进和优化的地方

2.2.1 除了以下八个CSS选择语法外Cherry实现了CSS3中全部的语法，

「：active」「：hover」」「：focus」「：visited」；「：first-line」「:first-letter」「：before」「：after」。首先最后两个并不是伪类选择器，而是伪元素，最后四个选择器都是没法通过JavaScript给其设置style属性的，因此也就没有实现的必要了，而对于前面四个CSS中的选择器，暂时还无法实现，jQuery貌似也没有实现这些选择器。

而cssQuery对CSS3的选择器支持范围还比较小，仅支持部分CSS3选择器，而不包括：nth-of-type、nth-last-of-type等伪类选择器，不包括CSS3中新增的属性选择器。因此Cherry比cssQuery对选择器的支持范围更广。

2.2.2 由于cssQuery是在2005年发布的，当时IE浏览器的最新版本还是IE6，（IE7是在2006年才发布），因此cssQuery需要支持很多古老的浏览器，当时浏览器对ECMAScript 5也都没有完全实现，很多DOM操作，事件监听等API在各个浏览器中实现也有很大差异。因此cssQuery需要对这个不同的浏览器进行兼容，而现在，十年过去了，浏览器基本都实现了ECMAScript 5，甚至很多浏览器也都实现了ECMAScript 2015中的大部分新功能。文档对象模型中的API也得到了基本统一。因此Cherry在实现浏览器兼容上可以更多的考虑下使用一些新的ECMAScript5中的新功能，并兼容IE8及以上的IE浏览器。这样就使得Cherry代码更加精简。

2.3 Cherry的实现原理

Cherry 的实现原理可以分为两步：第一步是对选择器字符串（也就是传入Cherry（）函数中的字符串参数）的处理，使得最后返回一个包含token和filter的数组，chopedArray。第二步就是沿着选择器字符串链从左右到有的方向（也就是chopArray中从0到chopArray.length-1），对DOM元素进行选择，最后返回所得结果，下面将分为这两步，详细论述其实现原理。

2.3.1 选择器字符串的处理

在这一小节中，为了便于理解，我将从一个具体的不规则的选择器入手，选择器字符串是怎样处理的，选择器字符串如下：

“ body > div#canvas .red , body div + .hidden ”

有面是一个group of selectors ，由两个selector 组成，通过逗号隔开的选择器字符串。

选择器字符串处理的第一步就是清除多余的空白：使得选择器变成如下：

“ body>div#canvas .red, body div+.hidden”

由于我们知道子孙选择器是一个空白符，因此在这一步中，我们要保证不能够把子孙元素选择器中的空白符清除掉，同时又要把多余的空白符清除掉。比如'>','+','~','(',')',','前后的空白符。这些空白符都不是必要的。

选择器字符串处理的第二个步骤就是添加在写选择器时省略的通配符（*），Universal selector为什么可以省略在W3C 选择器总结一节已经提到过，这儿就不赘述了。添加的目的就是为了在被切分为chopedArray的时候，正好是一个token一个filter的顺序。上面的字符串被处理后变成如下：

“ body>div#canvas *.red, body div+*.hidden”

上面两个处理步骤是通过parseSelector函数来完成了。函数如下：

```
var WHITESPACE = /\s*([\s>=+~(),]|^|$)\s*/g;
 var ADD_STAR = /([\s>+~,]|^)([:#.@])/g;
 function parseSelector(selector){
        return selector
            .replace(WHITESPACE, '$1')
            .replace(ADD_STAR, '$1*$2');
 }
```

其中WHITESPACE正则表达式的作用就是用来替换掉不必要的空白符，而ADD_STAR的作用就是在需要的时候添加通配符（*）。

字符串处理的第三部就是通过逗号把Group of selector 分成不同的selector，这一步比较简单，就不贴代码了。上面的选择器将被处理成如下两个选择器：

“ body>div#canvas *.red” “ body div+*.hidden”

字符串处理的最后一步，也是最关键的一步，就是把单个的selector，切分为token、filter、argument为元素的数组。上面的第一个选择器将被切分成如下数组。

[‘ ’, ’body’, ’>’, ’div’, ‘#’, ‘canvas’, ‘*’, ‘.’, ‘red’]

如果选择器中包含结构性伪类选择器，那么被切分出来的数组还包含argument元素。比如’ *:lang(en)’ 将被切分为如下数组：

[‘ ’, ‘*’, ‘:’, ‘lang’, ‘(’, ‘en’, ‘)’]

其中en就是argument。在进行切分之前，women还需要对每个选择器前面加一个空白符，也就是第一个token，表示初始选择document.documentElement的所有元素。那么什么是token，什么是filter呢?

简单来说，token就是表明这个simple selector是个什么类型的选择器，一次token可以是如下字符：' ‘ , ‘#’, ' .’, ' :’, ‘~' ,’+', ‘>' , ’@’。其分别表示子孙选择器、ID选择器、类选择器、伪类选择器、后面兄弟元素选择器、相邻元素选择器、子选择器、属性选择器。（@为什么表示属性选择器后面在属性选择器实现原理中会说到）。而filter就是指这些token后面用于筛选元素的字符，比如ID的值，className值等。因此在没有argument的情况下，一个chopArray就是一个token间隔一个filter的数组，而且token在filter前面。在Cherry中用于切分选择器的函数是chop（）。代码如下:

```
var STANDAR_SELECT = /^[^\s>~+]/;
 var CHOP = /[\s>+~#.:()@]|[^\s>+~#.:()@]+/g;
 function chop(selector){
        if( STANDAR_SELECT.test(selector) ) selector = '' + selector;
        return selector.match(CHOP);
 }
```

2.3.2 利用chopedArray 和 select 函数来选择DOM元素

首选看看select函数。

```
function select($from, $token, $filter, $arguments){
   var results = [];
   if($selectors[$token]){
     $selectors[$token](results, $from, $filter, $arguments);
   }
   return results;
 }
```

select 函数的作用就是根据token和filter组合成的简单选择器，从from中选择我们需要的元素，当然如果有arguments传入的话，我们还需要考虑argument的情况。我们从选择器字符串从左到右，逐步传入数组中的token，filter 和 argument。通过select函数获取

一个DOM元素数组，然后有把这个获取到的DOM元素数组作为from参数传入select函数中，继续进行选择，直到遍历完整个chopedArray。得到我们所需要的结果。其中$selectors是一个对象，里面包含以token为键，以选择方法为值的键值对。因此在$selecotrs中，每一个token都有一个专属的选择方法。代码在59到142行。$selectors对象中的方法当传入from，filter，arguments的时候，就会返回特定的DOM元素。

整个Cherry的实现原理就是上面这样的。下面将论述$selector对象中属性选择器和伪类选择器方法的实现。

2.3.3 属性选择器的实现原理

什么属性选择器，通过本章第一部分的介绍也比较清晰了。在实现属性选择器过程中，首先在$selectors对象中有一个关于属性选择器的方法。如下：

```
$selectors['@'] = function($result, $from, $attributeId){
   var test = null;
   if(attributeSelectors[$attributeId]) test = attributeSelectors[$attributeId].test;
   for(var i = 0, len = $from.length; i < len; i ++){
     if(test($from[i])) $result.push($from[i]);
   }
 }
```

属性选择器是通过「@」来标识的，而@符号也是在parseSelector阶段添加上去的，这在后面提高，@符号就是属性选择器的token值，在上面的方法中，我们只要传入备选的元素$from、属性选择器的ID值$attributeId值，然后调用attributeSelectors[$attributeId]上面的test方法，就能够判断备选的元素是否是我们选择的元素了。那么属性选择器的中点就是attributeSelectors这个数组对象了。

声明了两个变量，分别指向一个数组和一个对象。

```
var attributeSelectors = [];
var AttributeSelector = {match: /\[([\w-]+(\|[\w-]+)?)\s*(\W?=)?\s*([^\]]*)\]/};
```

attributeSelectors数组用来存放选择器中的属性选择器，其数组元素是通过AttributeSelector.create方法返回的属性选择器对象。而AttributeSelector对象则是用来存

放和属性选择器有关的方法，比如create方法，getAttrId方法， parse 方法。同时AttributeSelector对象中还有一个tests对象，用来元素是否符合属性选择器做出检测。下面将分别论述这些方法和对象。

```
AttributeSelector.create = function($propertyName, $compare, $value){
   var attributeSelector = {}, test;
   attributeSelector.id = this.PREFIX + attributeSelectors.length;
   attributeSelector.name = $propertyName;
   test = this.tests[$compare];
   attributeSelector.test = function(ele){
     return test? test(ele.getAttribute($propertyName),  $value.slice(1,-1)) : false;
   }
   return attributeSelector;
 };
```

该方法的作用就是创造一个attributeSelector对象，为其添加一个唯一的id值，和一个name属性，name属性就是属性选择器的属性名，比如[type ~= 'value']这个属性选择器。创造出来的attributeSelector对象的id值为「@」符号加上attributeSelectors的length。其name属性的值为type，上述方法中的$comparre参数就是「~=」，而$value就是选择器中type的值「value」。同时我们为attributeSelector对象赋予了一个test方法，用来检测传入的元素是否满足这个属性选择器。

在getAttrId函数中，我们会返回create函数创造出来的attributeSelector对象的id值，同时把这个attributeSelector对象推入attributeSelectors数组中储存起来。

AttributeSelector对象上最后一个方法是parse方法，顾名思义这个方法是对selector字符串进行解析的。其作用是把属性选择器切分成不同组成部分，包括nameSpace，compare，properName， value。并把这些切分好的组成部分传入getAttrId函数中，获取到与这个属性选择器相关的唯一ID值，然后把选择器中这个属性选择器替换为这个ID值。代码如下：

```
AttributeSelector.parse = function($selector){
```

```
  var matches, attrId;
  $selector = $selector.replace(this.NAMESPACE, '|');
  matches = $selector.match(this.match);
  if(matches)
    attrId = this.getAttrId(matches[0], matches[1], matches[2], matches[3], matches[4]);
  return $selector = $selector.replace(this.match, attrId);
 };
```

也就是说一个属性选择器如[type = 'test']经过上面三个方法后，将被解析成'@id',其中id 值就是该属性选择器对应的attributeSelector在attributeSelectors中的索引。而在attributeSelector中就有一个test方法，可以检测传入的元素是否匹配该属性选择器。以上就是整个属性选择器实现的原理，对于parse方法怎么添加到parseSelector方法上，使得一开始就实现属性选择器的替换，也就是对parseSelector方法进行重写。具体做法将在第三种所遇到的问题中论述。

2.3.4 伪类选择器的实现原理

伪类选择器和属性选择器一样，在Cherry中占有相当大的代码量。首先伪类选择器在选择器对象$selectors中的键为「: 」，也就是「: 」就是伪类选择器的token值。「: 」后面的字符串就是伪类选择器的filter，比如root选择根元素，empty选择不包含子元素的元素。nth-child（2）选择其父元素中的第二个子元素。last-child选择器父元素的最后一个子元素等等。也就是说filter能够筛选我们的伪类选择器选择的元素。伪类选择器在$selectors对象中的方法如下：

```
$selectors[':'] = function($result, $from, $pseudoClass, $arguments){
  var $test = null;
  if(pseudoClasses[$pseudoClass]) $test = pseudoClasses[$pseudoClass];
  if($pseudoClass == 'root') $from.push(document.documentElement);
  for(var i = 0, len = $from.length; i < len; i++ ){
    if($test($from[i], $arguments)) $result.push($from[i]);
  }
  return $result;
 };
```

$result参数是该方法需要返回的结果，$from是所有备选元素，$pseudoClass就是伪类选择器的filter，$arguments就是伪类选择器的参数，如nth-child（2）中括号里面的2.
当把这些参数传入伪类选择器方法中，就能够从$from中取得被选择的元素，然后存入$result数组中作为函数的返回值。
当然伪类选择器的难点也就是pseudoClasses对象了。首先我们需要的全局声明一个pseudoClasses对象，用来存放不同的伪类选择器。键名就是伪类选择器中的filter。
比如pseudoClasses['link'] = function($element, $arguments){}
pseudoClasses['lang'] = function($element, $arguments){}
…
对于不同的伪类选择器filter，都有一个与之相对应的选择器方法，当传入备选元素后我们就能够对备选元素是否是我们选择的元素做出判断，如果是我们选择的元素，就返回ture，否则返回false。当然伪类选择器中也遇到了问题，就是对：nth-last-of-type（）、nth-of-type（）、nth-child（）、nth-last-child（）四个伪类选择器在pseudoClasses对象中对应的方法实现上遇到了问题。将在第三章论述。
第二章结束，以上就是整个Cherry选择器引擎的实现原理。

三、实现Cherry 选择器引擎过程中遇到的问题及解决方案

在本章主要描述在实现Cherry 选择器引擎过程中遇到的问题集解决方案，首先是问题描述，然后是解决方案。

3.1 DOM操作中遇到的问题

虽然现在很多浏览器都支持previousElementSibling，nextElementSibling、children等元素节点API,但是依然有些浏览器对这些API的支持程度还不高，比如IE8就不支持

children属性，同时现在浏览器还没有获取所有子孙元素的API,或者获取某个元素后面的所有兄弟元素的API，而这些API在Cherry中都是需要的，因此这些兼容性问题、这些需求摆在面前都使得需要自己实现这些DOM操作方法。

在Cherry中实现的access property 包括：

isElementNode：用来判断一个元素是否是元素节点

isTextNode： 用来判断一个节点是否是文本节点

prevEleSibling： 用来获取元素前面相邻的兄弟元素节点

nextEleSibling： 用来获取元素后面的相邻的兄弟元素节点

afterEleSiblings： 用来获取元素后面的所有兄弟元素节点，返回一个元素数组

firstEleChild： 用来获取某元素的第一个子元素

lastEleChild： 用来获取某元素的最后一个子元素

childrenEle：用来获取某元素的所有子元素

childEleCount： 用来获取某元素的所有子元素个数

desEleNodes： 用来获取某元素的所有子孙元素

最开始的想法是把上面的方法写成简单的函数，比如：

function isElementNode（element）{}

但是这些简单函数又不能够链式调用，于是我就想到了把这些方法写到Node或者HTMLElement的原型上面，这样就可以实现链式调用了，比如：

```
Node.prototype.isElementNode = function(){
  var ele = this;
   …
}
```

但是这样的话和原生的属性还是不同的，毕竟这样写，isElementNode还是方法，调用的时候还是需要写小括号，既然不传参数进去了为什么不把括号也省了呢？于是我就想到了在Node或者HTMLElement对象的原型上定义access property。也就是getter属性。实现过程如下：

```
Node.addGetter = function(name, func){
    Object.defineProperty(this.prototype, name, {
```

```
            configuable : true,
            enumerable  : false,
            get         : function(){
                return func(this);
            }
        });
        return this;
    };
```

首先我在Node构造函数上面添加一个addGetter的静态方法，起作用就是向其原型上面添加getter属性，属性名就是传进addGetter方法的第一个参数，返回值就是传进addGetter函数的第二个参数函数的返回值。

在上面这个函数中，应该注意到两次出现了this对象，而且两次的this对象并不是指代的同一个对象。在第一个this对象（this.prototype）中，this对象指代的是Node构造函数。在第二个this对象中（return func（this））这儿的this对象指代的是调用addGetter方法的Node实例或者HTMLElement实例。

通过如下函数定义isElementNode getter属性。

```
Node.addGetter('isElementNode', function(self){
    return !!(self && self.nodeType == 1);
});
```

我们只需要判断self的nodeType是否为1，就能够判断某个元素是否是元素节点。

通过addGetter方法定义nextEleSibling属性。

```
Node.addGetter('nextEleSibling', function(self){
    if(self.nextElementSibling) return self.nextElementSibling;
    var element = self;
    while(element && (element = element.nextSibling) && !element.isElementNode)
continue;
    return element;
```

```
});
```

在上面代码中，首先会判断self是否已经具有了nextElementSibling方法，如果有了就返回该方法，如果没有就自己定义该方法。

DOM操作中最复杂的getter属性应该就是desEleNodes了，用来获取所有子孙元素。

```
HTMLElement.addGetter('desEleNodes', function(self){
  var elements = [];
  (function collectElements(elems){
    Array.prototype.push.apply(elements, elems);
    for(var i = 0; i < elems.length; i++){
      if(elems[i].childrenEle){
        collectElements(elems[i].childrenEle);
      }
    }
  })(self.childrenEle);
  return elements;
});
```

该属性添加到HTMLElement的原型上面，该方法使用了JavaScript中的递归和自调用函数。也就是说我们判断一个子元素中如果还有子元素，我们就继续调用collectElements方法。同时该添加属性的方法上还用到了一个在类数组上使用数组方法的技巧，就是通过apply。

```
Array.prototype.push.apply(elements, elems);
```

因为elements可能是类数组，类数组上面没有push方法，因此需要通过apply来在elements上调用push方法。

以上就是我在实现DOM操作上面遇到的问题。

3.2 说说monkey patching

Monkey Patching： 其含义就是首先获取某一对象方法的引用，然后重新定义新的方法，在定义新的方法的时候，有选择性的调用原有方法。猴子补丁通常用来为已有方法追加新的功能，或者完全重写原有方法，但是不推荐完全重写，而且在做猴子不

定的时候，一定需要在文档中注明，避免别人在使用被打过猴子补丁的类库中的方法时，出现错误。

在Cherry中，定义了一个parseSelector 的方法函数，用来对传入的选择器字符串进行一些清除空白和添加通配符的操作。但是这个方法并不能够选出并标示属性选择器，也就是说这个方法并不能够处理属性选择器的选择器字符串。不能够把选择器中的属性选择器替换为'@id'，而在select 函数中我们使用的就是属性选择器的id值，因此在不改变parseSelector函数的情况下，追加性的功能，就需要用到猴子补丁的方法。在Cherry中是这样实现的。

```
//monkey-patch rewrite parseSelector
 var _parseSelector = parseSelector;
 parseSelector = function($selector){
  return _parseSelector(AttributeSelector.parse($selector));
 };
```

首先我们把对parseSelector方法的引用赋值给_parseSelector变量，保留对原方法的引用，然后开始重写parseSelector方法，在重写的过程中又调用了原来的parseSelector方法，也就是在原来对选择器字符串处理前添加了一个属性选择器的替换工作。不如选择器：

input[type='color']

将被处理成如下形式：

input@id

其中id值是该属性选择器在attributeSelectors中的索引。

3.3 :root 伪类选择器的问题

root伪类选择器是CSS3引入的新的选择器语法，其作用是用来选择根元素，也就是等同于document.documentElement元素，在使用过程中应该注意root选择器前面不能够使用其他选择器语法，正确做法如下：

： root

错误示范如下：

p:root

因为在实现root伪类选择器时，选择的是document.documentElement元素，但是我们在对选择器字符串进行切分时，也就是在调用chop函数时，会给选择器前面添加一个空白符，「’ ‘」, 比如上面的选择器「:root」,经过parseSelector和chop函数处理后，返回如下数组。

[“ ”, “*”, “:”, “root”]

然后再调用select函数是，首先就会选取document.documentElement 的所有子孙元素，然后再在这些备选元素中选择，我们需要的root元素，当然这样是不行的，因为document.documentElement的子孙元素中不包含自己（document.documentElement)。因此也就无法在其子孙元素中找到我们需要的元素了，我的解决方案比较简单，就是在$selecotrs[‘:’] 方法中，如果伪类的filter是root。我就在其传入的$from中加入一个

document.documentElement元素，实现代码如下：

```
$selectors[':'] = function($result, $from, $pseudoClass, $arguments){
   var $test = null;
   if(pseudoClasses[$pseudoClass]) $test = pseudoClasses[$pseudoClass];
   if($pseudoClass == 'root') $from.push(document.documentElement);
   for(var i = 0, len = $from.length; i < len; i++ ){
     if($test($from[i], $arguments)) $result.push($from[i]);
   }
   return $result;
 };
```

主要起作用的就是下面这句代码：

```
  if($pseudoClass == 'root') $from.push(document.documentElement);
```

3.4 nth-child\nth-last-child\nth-of-type\nth-last-of-type这个四伪类选择器的实现

这四个伪类选择器有一个相同点，就是传入的参数类型相同，可以是如下一些形式：

n： 任意非负整数

even：偶数

odd：奇数

an + b： 其中a、b是非负整数，（当a = 1，b = 0 ，其实就是n了 ，当 a = 0, b = 正整数， 其实也就是选择b）

为什么说实现这四个伪类选择器比较困难了，首先这四个伪类选择器传入的参数形式就比较多样，其次伪类选择器的filter中包含last和不包含last，又代表了是否需要逆向索引还是正向索引。（不包含last是正向索引）

在Cherry中，我定义了一个nthChild函数，这个函数应该是Cherry中最为复杂的一个函数了，其用来处理上面遇到的所有问题，函数如下：

```
function nthChild($element, $argument, $travese, type){
  var multi, step, children = [], tagName = $element.tagName.toLowerCase();
  switch($argument){
   case 'n': return true; break;
   case 'even': $argument = '2n'; break;
   case 'odd' : $argument = '2n+1'; break;
  }
  if(type && type == 'nthType'){
   Array.prototype.forEach.call( $element.parentNode.childrenEle, function(ele){
    if(ele.tagName.toLowerCase() === $element.tagName.toLowerCase())
     children.push(ele);
   });
  } else children = $element.parentNode.childrenEle;
  function _checkIndex($index){
   $index =  Object.prototype.toString.call($travese).indexOf('Function') &&
        $travese === 'prevEleSibling' ? children.length - $index : $index -1;
   return children[$index] === $element;
  }
```

```
if(!isNaN($argument))
  return _checkIndex($argument);
$argument = $argument.split('n');
multi = parseInt($argument[0]);
step = parseInt($argument[1]);
if(isNaN(step)) step = 0;
if(isNaN(multi)) multi = 1;
if(multi == 0) return _checkIndex(step);
var count = 1;
if(type && type == 'nthType') {
  while($element = $element[$travese]) {
    if($element.tagName.toLowerCase() == tagName) count++;
  }
} else {
  while($element = $element[$travese]) count++;
}

if(multi == 1)
  return children.length - count + 1 >= step;
return count % multi === step % multi && children.length - count + 1 >= step;
}
```

nthChild的作用就是用来传入备选元素、伪类选择器的参数、伪类选择器的索引方向和伪类选择器的类型这四个参数，然后判断备选元素是否是我们所选择的元素。

传入的四个参数含义：

$element: 该参数是传入的备选元素，如果函数返回的结果为true，那么该元素就是需要的元素，如果函数返回结果为false，该元素就不是所选元素。

$argument:该参数是上面四个伪类选择器小括号中的参数，如 n， 2n + 1等。

$travese: 其实这是HTMLElement实例上的两个getter 属性，可以有两个取值，nextEleSibling和PrevEleSibling，起作用分别是用来选择相邻的后面的兄弟元素和选择

相邻的前面的兄弟元素。在这个函数中，nextEleSibling表示正向索引，而PrevEleSibling表示反向索引。

type: type是伪类选择器的类型，当不传入type值时，表示nth-child和nth-last-child两个伪类选择器，当传入「type」值是，表示nth-last-of-type和nth-of-type两种伪类选择器。

该函数中有一个闭包函数，

```
function _checkIndex($index){
    $index =  Object.prototype.toString.call($travese).indexOf('Function') &&
        $travese === 'prevEleSibling' ? children.length - $index : $index -1;
    return children[$index] === $element;
  }
```

其作用是用来给定指定的$index判断其是否是所选的元素。

而nthChild的其他部分就是处理不同形式的伪类选择器参数了。

3.5 Cherry 的性能问题

Cherry 和浏览器选择器引擎或者sizzle选择器引擎的解析顺序是不同的，后者都是选择的从右到左的解析顺序，这也是为什么很多CSS大师都建议我们在写选择器的时候，最右边的key Selector尽量写具体一点，这样在第一步就能够大大缩小搜索范围。

而Cherry的解析顺序和传统的选择器引擎解析顺序相反，其更像人们的书写选择器的顺序，从左到右，这也使得在使用Cherry的过程中，尽量把选择器链左边的选择器写得更加具体一些。

当然我之所以选择从左到右的解析顺序，并不是我觉得从右到左的解析顺序就不好，反而只是为自己提供另一种思路而已。在第四部分，以后的优化和改进一章将进一步讨论如何选择解析方向来提高选择器的性能问题。

Cherry 的另一个性能问题的提升就是Cherry的记忆功能，在Cherry代码中，定义了一个cache 对象，用来缓存选择器和已经选择到的元素。其实现代码在main函数中如下：

```
if($useCache && cache[cacheSelector]){
```

```
 from = cache[cacheSelector];
} else {
 from = select(from, token, filter, arguments);
 if($useCache) cache[cacheSelector] = from;
}
```

首先判断该选择器是否已经被缓存，如果缓存中有，我们就直接使用缓存的选择器所选择的元素，如果没有缓存，我就调用select方法，获取所选元素，然后将其缓存在cache对象中。这样大大提高了选择器引擎的效率。默认情况下，在使用Cherry都是缓存的。

四、将来对Cherry的优化和改进

4.1 在2015年六月，ECMAScript2015发布，在新的JavaScript标准中，定义了新的数据结构Set，以及其他一些新的API，在以后学习ECMAScript2015并熟悉后，且到时候浏览器也基本支持ECMAScript2015。可以通过ECMAScript2015对Cherry进行改写。比如就可以使用到Set数据结构，因为Set中没有重复的元素，这个可以用来很好的存取

我们的选择器方法等，或者用来缓存已经选择的元素，并且Set能够和数组相互转换，在Set上求交集和并集也是很方便的。

4.2 HTML 5中引入了worker API，这也是提高Cherry性能的福音，worker能够允许两个或多个JavaScript文件同时工作，也就是说可以建立多个worker同时来对选择器字符串来进行解析，那么可不可以这样做呢？生成两个worker，一个worker从字符串左边向右解析，而另一个worker从右向左解析，两个worker在字符串中间汇合，然后把两个worker选择到的元素通过Set存储，然后在对Set进行求交集，这样对于比较长的选择器是否效率更高呢？

五、Reference

1、Selectors Level 3 W3C Recommendation 29 September 2011 http://www.w3.org/TR/css3-selectors/

2、HTML5 Canvs 核心技术 图形、动画与游戏开发 David Geary

3、精通 JavaScript John Resig

4、Dean Edwards的个人网站http://dean.edwards.name/my/cssQuery/

5、JavaScript 权威指南 David Flanagan

6、HTML5 高级程序设计 Peter Lubbers、Brain Albers、Frank Salim

7、jQuery的选择器引擎sizzle（ http://sizzlejs.com/）