CSS Selector Complexity module

Introduction

This document has as purpose to expose the degree of complexity of a CSS selector. Many web developers query selectors that are not possible to implements due to a too great time needed to evaluate them. This document will expose some ways to evaluate the complexity of a selector. In fact, there's not only one way to define the complexity of a selector. This document can't talk about all kind of complexity, but it will be as accurate as possible.

Loop complexity

Simple loop complexity

Explanation

Each CSS rule need to loop to find elements that can or not match a rule. Loops are things that take the longest time during the matching step of a rule. The most common type of loop is a search through all children of an element of the document.

Samples of simple loop complexity

"span" is converted in :

```
      Dim List1 as new List(Of DOMElement)
      For each el1 as DOMElement in document.all
            If el1.tagName == "span" Then
                  List1.add(el1)
            End if
      Next el1
      Return List1
```

There's only one loop for this selector

"ul.list li" is converted into :

```
      Dim List1 as new List(Of DOMElement)
      For each el1 as DOMElement in document.all
            If
                  el1.tagName == "ul"
                  AndAlso el1.getAttribute("class") == "list"
            Then
                  For each el2 as DOMElement in el1.all
                        If el2.tagName == "li" Then
                              List1.add(el2)
                        End if
                  Next el2
            End if
      Next el
```

There are two loops for this selector.

Referred loop complexity

Explanation

The previous model is a good way to represent simple selectors but it can be used when we talk about more complex selectors. Actually, each loop take less time that the previous one because we walk a smaller part of the DOM. The reduction of the number of walked element can be found by considering the number of chance the parent selector match a lot of element or not. In fact, this is not possible to know such information without having a document on which we can rely. So, the only way to determine if a selector matches a lot of things or not is to use the "priority" of the selector. More the priority is high, less chance we have to have redundant elements.

Compute the priority

Priority in CSS isn't defined as a number but an array of number of time a type of "condition" appears. A selector containing an ID is always more specific that a selectors that contains numbers of classes. If we want to convert theses data's to a real number, we must decide a coefficient that's used to convert an ID in a certain number of classes, and a class in a certain number of tags …

I think (but it's only arbitrary) that the coefficient can be 10 because it's an easy number and that a selector containing ten classes is as specific as a selector that contains an ID. Now, we must simplify the way priority is computed by the UA because a lot of elements are taken in count. Here, we shall define three types of priority types: ID, Attributes and Tags.

- An ID = 10 attributes = 100 tags
- A class = 1 attribute = 10 tags

Here are some easy samples:

- "div#ads" ➔ (1 + 100) ➔ 101
- "div.ads" ➔ (1 + 10) ➔ 11
- "div span a" ➔ (1) + (1) + (1) ➔ 3
- "div.content span a[href^=http://]" ➔ (1 + 10) + (1) + (1 + 10) ➔ 23

Naturally, we must take in count some others operators like "+", ">". Again, it's not possible to find a perfect way to render these attributes, but we can consider the following as good approximations:

- "span > span" ➔ (1 + 25) + (1) ➔ 27
- "span + span" ➔ (1 + 1) + (1) ➔ 3
- "span ~ span" ➔ (1 + 1) + (1) ➔ 3
- "span:nth-child(3)" ➔ "*.-first-child + * + span" ➔ (0 + 10 + 1) + (0 + 1) + (0 + 1) + (1) ➔ 14
- "span:nth-child(1n)" ➔ "span.-1n-child" ➔ (1 + 10) ➔ 11
- "span:first-of-type" ➔(1 + 15) ➔ 16
- "span:only-child" ➔ (1 + 15) ➔ 16
- "a:href" or "input:active" ➔ (1 + 50) ➔ 51

Compute loop value

Now, we can evaluate the time needed for a loop to execute:

$$LoopValue = \frac{1}{PriorityOfParentSelector}$$

Samples of computed loop value

(We round the value at the second decimal place. Please note that the Priority of the document is "0.1")

- `L(a b c d e) = 1/0.1 + 1/1 + 1/2 + 1/3 + 1/4 = 12.08`
- `L(div#content) = 0` *(Arbitrary, an ID selector has LoopValue = 0)*
- `L(div#content span) = 0 + 1/101 = 0.01`
- `L(span div#content) = 1/0.1 + 1/1 + 11`
- `L(a:href) = 1/0.1 = 10.00`
- `L(a:href span) = 1/0.1 + 1/51 = 10.02`
- `L(*) = 20` *(Arbitrary, the "*" selector has LoopValue = 20)*
- `L(* span) = = 1/0.1 + 1/0.1 = 20`
- `L(span *) = 1/0.1 + 1/1 = 11`
- `L(span > *:hover) = 1/0.1 + 1/26 = 10.04`
- `L(* > span:hover) = 1/0.1 + 1/25 = 10.04`
- `L(:root) = 0` *(Arbitrary the ":root" selector has LoopValue = 0)*
- `L(:root a span) = L(*.root a span) = 1/0.1 + 1/11 = 10.1`
- `L(:root > body > div) = L(#root > body div) = 0 + 1/125 + 1/126 = 0.02`
  ":root" counts as "#root" only if there's only '>', '+' and/or '~' after them.*
  If not, it counts as ".root" and have less impact than "#root"*

UPDATE COMPLEXITY

Principe of update complexity

Explanation

As you can see the loop complexity is not sufficient to say if a rule is easy to calculate in a UA or not. Why? Because it takes in count only the time that's needed to find all elements that match the rule through the DOM. But in it doesn't take in count the "update complexity" or, if you prefer, the number of events that must be watched if we want to continue to serve the rule when the document changes (ECMAScript can add or remove elements of the DOM, add, remove or modify attributes, and the user make some pseudo classes like ":hover" and ":active" very variables). If the browser wants to avoid updating all rules when something changes in the document, it must look at specific changes that can have effect on a specific rule. In addition to take a lot of memory, the update handles that are too frequent are bad for the CSS Speed because they need to redo some part of the rule matching step (or the whole watching step, in some cases).

Sample

Please consider the following XML document:

```
<data>
      <author name="…" birthday="…" nativelang="…" … >
            <book title="…" cost="…" availabesince="…" lang="…" … />
            …
      </author>
      …
</data>
```

And this CSS Rule:

```
author[name*=king] book[title*=dead] { background: yellow; }
L(…) = 10.1
```

We must consider each time the rule can (un)match an element of the document.

The numbers between parentheses are the LoopValue of all loops that are again need and the number of conditions to expect before.

Document's listeners

- Each time an element is added to the DOM, anywhere in the document, we must check if it's an AUTHOR element. If yes, we should add listeners to his mutation's events and check if it matches the rule, and we must do the same for his children. (0.1, 1)

AUTHOR's listeners

- Each time the "name" attribute became modified in a hooked AUTHOR element, we must recheck if it matches the rule, and we must do the same for his children. (0.1, 1)
- Each time an element is added to the DOM into the AUTHOR element, we must check if it's a BOOK element. If yes, we should add listeners to his mutation's events and check if match the rule. (0.1, 1)

BOOK's hook

- Each time the "title" attribute became modified in a hooked BOOK element, we must recheck if it matches the rule. (0, 1)

Update complexity

User actions that must be hooked: 0

DOM Additions that must be hooked: (1) + (1) = 2

DOM Changes that must be hooked: (1) + (1) = 2

Computed update complexity

LoopValues are multiplied by 10 and conditions divided by 100 to make numbers more reliable

$$ListenerComplexity = 100 * (10 * LoopValues + 0.01 * Conditions)/PriorityOfElement$$

$$\begin{aligned} UpdateComplexity \\ &= 5(\sum UserActionListenerComplexity) \\ &+ 1(\sum DOMChangeListenerComplexity) \\ &+ 1(\sum DOMAdditionListenerComplexity) \end{aligned}$$

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- "AUTHOR[name]" hook : 100*(10*0.1 + 0.01*1)/11 = 9.18
- "BOOK[title]" hook : 100*(10*0 + 0.01*1)/22 = 0.45
- Total : 9.63

DOM Additions that must be hooked:

- "AUTHOR added in document" hook : 100*(10*0.1 + 0.01*1)/11 = 9.18
- "BOOK added in AUTHOR" hook : 100*(10*0 + 0.01*1)/22 = 0.45
- Total : 9.63

Total = 5*0 + 9.63 + 9.63 = 19.26

Value of update complexity

The value of the computed update complexity shows the additional time that you should add to your rule if the document changes. If you work with a static document, only User Actions are to take in count. As you can see, self a simple rule have a high update complexity. We always must take in count the update complexity when we considerate the possibility to use or add a selector to the CSS.

Ameliorate the update complexity

Another fact: The update complexity can be reduced by writing more specific selectors. It also make the LoopValue smaller so web developers should always use the most closest possible implementation of a selector. We shall do the same calculation as before, but for a more specific version of the previous selector: ":root > author[name*=king] > book[title*=dead]"

LoopValue = (0) + (1/125) + (1/136) = 0.01 *(It less small than the LoopValue of the previous one !)*

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- "AUTHOR[name]" hook : 100*(10*0.01 + 0.01*1)/125 = 0.09
- "BOOK[title]" hook : 100*(10*0 + 0.01*1)/136 = 0.01
- Total : 0.1

DOM Additions that must be hooked:

- "AUTHOR added in document" hook : 100*(10*0.01 + 0.01*1)/125 = 0.09
- "BOOK added in AUTHOR" hook : 100*(10*0 + 0.01*1)/136 = 0.01
- Total : 0.1

Total = 5*0 + 0.1 + 0.1 = 0.2 *(Do we need to compare with the other one?)*

THE WAY TO KNOW IF A SELECTOR IS QUICK OR NOT

Please note that this is only an indicative value, not the reality! It can fails for some special cases

For selectors

The first thing we need to know before saying if a selector is quick or not:

- A pseudo-code that explains the way the matching elements are found.
- The LoopValue and UpdateValue of "selector"

For operators

The first thing we need to know before saying if an operator is quick or not:

- A pseudo-code that explains the way the matching elements are found.
- The LoopValue and UpdateValue of "a {OP} b"

For pseudo-classes

The first thing we need to know before saying if a pseudo-class is quick or not:

- The LoopValue and UpdateValue of "a:pseudo"

1e sample (:hover)

Evaluated selector is: “a:hover”

User actions that must be hooked:

- Mouseover/mouseout on “a” : 100/0.1*(10*0.01 + 0.01*1) = 20
- Total: 20

DOM Changes that must be hooked:

- Total : 0

DOM Additions that must be hooked:

- “A added in document” hook : 100/0.1*(10*0 + 0.01*1) = 10
- Total : 10

Total = 5*20 + 0 + 10 = 110

2e sample (:only-child)

Evaluated selector is: “a:only-child”

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- Total: 0

DOM Additions that must be hooked:

- “A added in document” hook : 100/0.1*(10*0 + 0.01*1) = 10
- “* added anywhere in the document”: 100/0.1*(10*0+0.01*1) = 10
- “* removed anywhere in the document”: 100/0.1*(10*0+0.01*1) = 10
- Total: 30

Total: 30

3e sample (default operator)

Evaluated selector is: “a *”

LoopValue = (1/0.1) + (1/1) = 11

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- Total: 0

DOM Additions that must be hooked:

- “A added in document” hook : 100/0.1*(10*1 + 0.01*0) = 100
- “* added anywhere in A”: 100/1*(10*1+0.01*0) = 1000
- “* removed anywhere in A”: 100/1*(10*1+0.01*0) = 1000
- Total: 2100

Total: 2100

4e sample (:matches)

Evaluated selector is: “a:matches(*)”

Exists is supposed to return true if a “” element exists in the anchor element, false otherwise.*

LoopValue = (1/0.1) + (1/1) = 11

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- Total: 0

DOM Additions that must be hooked:

- “A added in document” hook : 100/0.1*(10*1 + 0.01*0) = 100
- “* added anywhere in A”: 100/1*(10*1+0.01*0) = 1000
- “* removed anywhere in A”: 100/1*(10*1+0.01*0) = 1000
- Total: 2100

Total: 2100

5e sample (:exists)

Evaluated selector : “a:exists(*)”

Exists is supposed to return true if a “” element exists in the document, false otherwise.*

LoopValue : (1/0.1) + (1/0.1) = 20

User actions that must be hooked:

- Total: 0

DOM Changes that must be hooked:

- Total: 0

DOM Additions that must be hooked:

- “A added in document” hook : 100/0.1*(10*0 + 0.01*0) = 0
- “* added anywhere in document”: 100/0.1*(10*10+0.01*0) = 10000
- “* removed anywhere in A”: 100/0.1*(10*10+0.01*0) = 10000
- Total: 20000

Total: 20000