Style Invalidation in Blink

rune@opera.com

DOM Mutations and Computed Style Changes

Mutating the DOM, either by adding or removing nodes, or modifying attributes and input states, will change the computed style for a set of elements in the DOM. The reason is that the evaluation of selectors will change due to these mutations. Modifying the tree structure will make selectors change evaluation because of combinators and structural pseudo classes, and modifying attributes and input state will change evaluation because of pseudo classes, id selectors, class selectors, and attribute selectors. The tree structure changes will cause render tree reattachment, hence full style recalc, for the removed/inserted node and its whole subtree. That type of changes will also cause recalculation of computed style for siblings of the inserted/removed node because of adjacent combinators and structural pseudo classes affecting those siblings. The rest of this text will focus on changes that do not modify the tree structure.

The number of nodes that are affected by a DOM mutation depends on the selectors present in author and UA stylesheets for the document. With the currently implemented CSS selectors in Blink, the elements which can possibly have their computed style affected by a change in a given element are:

- The element itself
- Its descendants
- All succeeding siblings and their descendants

The worst case can be expressed with this rule:

```
.a, .a *, .a ~ *, .a ~ * * { }
```

When setting the class attribute to “a” on an element, the first selector will select the element itself, the second all its descendants, the third all its siblings, and the fourth all sibling descendants. Consequently, you have to recalculate the computed style for all those descendants and siblings. The common case, however, when changing a class, id, or other attribute on an element, a computed style recalc is necessary for far fewer elements. Take this example:

```
.a .b {}
```

Only descendants of the element are affected when setting class=”a”, and only elements which have a class attribute that contains the class “b”.

Performance Considerations

The brute force approach to computed style recalculation, given the possible set of affected siblings and descendants in the previous section, is to recalculate all elements in the worst case scenario. That requires no knowledge about the CSS selectors that apply in a given document.

At the beginning of 2014 that was pretty close to what the Blink implementation did. The meta-data stored were hash sets where an id, class, or attribute were in a hash set if it occurred in a selector, if an element could possibly match a selector containing an adjacent combinator, and a document-wide maximum number of consecutive adjacent combinators. In practice, that meant we could skip sibling subtree recalculation in most cases, but all descendants of the modified element always got their computed styles recalculated. For instance, that meant changing anything on the BODY element caused a full document recalc.

The performance goal of the Blink project is to be able to run web content at 60fps on a mobile phone, which means we have 16ms per frame to handle input, execute scripts, and execute the rendering pipeline for changes done by scripts through style recalculation, render tree building, layout, compositing, painting, and pushing changes to the graphics hardware. So style recalculation can only use a fraction of those 16ms. In order to reach that goal, the brute force solution is not good enough.

At the other end of the scale, you can minimize the number of elements having their style recalculated by storing the set of selectors that will change evaluation for each possible attribute and state change and recalculate computed style for each element that matches at least one of those selectors against the set of the descendants and the sibling forest.

At the time of writing, roughly 50% of the time used to calculate the computed style for an element is used to match selectors, and the other half of the time is used for constructing the RenderStyle (computed style representation) from the matched rules. Matching selectors to figure out exactly which elements need to have the style recalculated and then do a full match is probably too expensive too.

We landed on using what we call descendant invalidation sets to store meta-data about selectors and use those sets in a process called style invalidation to decide which elements need to have their computed styles recalculated.

Descendant Invalidation Sets

Definitions

(1) *Given an element E, and a property value P that is changed on E. A descendant F of E needs to have its style recalculated if there is a descendant invalidation set S for P, and there is a property value on F that is a member of S. Examples of P are class names, id attribute values, and names of other attributes.*

(2) *An empty invalidation set means that the computed style only needs to be recalculated for the element for which the property value was changed.*

(3) *There is a boolean wholeSubtreeInvalid flag associated with each invalidation set which is true if all descendants need to have their computed styles recalculated.*

(4) *There is a boolean treeBoundaryCrossing flag associated with each invalidation set which is true if the invalidation set invalidates elements in shadow trees.*

(5) *There is a boolean insertionPointCrossing flag associated with each invalidation set which is true if the invalidation set invalidates elements distributed under content element descendants.*

Constructing Invalidation Sets

Invalidation sets are constructed and aggregated by iterating through all CSS selectors. For each selector we go through a two-step process where we first extract properties from the simple selectors of the rightmost compound selector which we then add as properties to the invalidation sets for properties found by looking at simple selectors in the other compound selectors.

The simple selectors from rightmost compound selector are the interesting properties to be added to the invalidation sets since those selectors are the ones that will match the element that will get the declarations from the style rule. For instance, the rule:

```
.a .b { color:green }
```

will contribute to the computed style of elements with class b while elements with class a will just take part in the selector matching.

There are some simple selectors which are currently skipped and not added to the invalidation sets. One example is negated selectors. It is not impossible to implement, but we currently do not support negated members of invalidation sets.

The algorithm per selector can be described as:

(1) *For each simple selector in the rightmost compound selector that maps to an invalidation set property P:*
    (a) *Add P to a temporary set S.*
    (b) *If there is no descendant invalidation set for P, create an empty invalidation set for P.*
(2) *For all other compound selectors*
    (a) *For each simple selector that maps to an invalidation set property P*

(i) *If there is no invalidation set for P, create an invalidation set for P.*
(ii) *If S is empty, or if the combinator right of the compound selector is an adjacent combinator, set the wholeSubtreeInvalid flag on the invalidation set for P.*
(iii) *Otherwise add all members of S to the set for P.*
(iv) *If there is a ::content pseudo element anywhere right of the simple selector for P, or the simple selector for P is in a :host or :host-context pseudo class, set the insertionPointCrossing flag on the set for P.*
(v) *If there is a ::shadow or /deep/ combinator anywhere right of the simple selector for P, or the simple selector for P is in a :host or :host-context pseudo class, set the treeBoundaryCrossing flag on the set for P.*

There is another complexity to P extraction from the rightmost compound selector when you have pseudo classes which takes a compound selector list like :-webkit-any(). These selector lists are disjunctions in the sense that only one compound selector needs to match in order to match the whole pseudo. Consequently, -webkit-any(*, .a) is a universal selector when present in the rightmost compound as far as the invalidation sets are concerned. Yet, when present in other compound selectors, Ps from all selector list compounds need to have their invalidation sets constructed. See the Examples section.

Notation

In the examples and text below, we use the following notation:

```
a - element name
.a - class name
#a - id
[a] - attribute name
* - wholeSubtreeInvalid
! - treeBoundaryCrossing
!! - insertionPointCrossing

P { P0, P1, .. , PN } - invalidation set for P
```

Examples

1. Selectors:

   ```
   .a { }
   ```

   Invalidation sets:

   ```
   .a { }
   ```

2. Selectors:

   ```
   .a .b { }
   .c { }
   ```

   Invalidation sets:

   ```
   .a { .b }
   .b { }
   .c { }
   ```

3. Selectors (:not(.b) does not map to a P, so there are no Ps in the rightmost compound which causes the wholeSubtreeInvalid to be set for the set for ".a"):

   ```
   #x * { }
   #x .a { }
   .a :not(.b) { }
   ```

   Invalidation sets:

   ```
   .a { * }
   #x { *, .a }
   ```

4. Selectors (treeBoundaryCrossing and insertionPointCrossing):

   ```
   :host-context(.a) span { }
   :host(.b) input[disabled] { }
   [type] ::content > .c { }
   ```

   Invalidation sets:

   ```
   .a { span, !, !! }
   .b { input, [disabled], ! }
   .c { }
   [type] { .c, !! }
   ```

5. Selectors (disjoint selector lists):

   ```
   .a :-webkit-any(:hover, .b) { }
   .c :-webkit-any(.d, .e) { }
   :-webkit-any(.f, .g) { }
   :-webkit-any(.h, *) #i { }
   ```

Invalidation sets:

```
.a { * }
.b { }
.c { .d, .e }
.d { }
.e { }
.h { #i }
```

6. Selectors (negated simple selectors mapping to properties in non-rightmost compound selectors):

   ```
   :not(.a):not(.b) > span#id { }
   ```

   Invalidation sets:

   ```
   .a { span, #id }
   .b { span, #id }
   ```

7. Selectors (adjacent selectors causing wholeSubtreeInvalid):

   ```
   .a + .b #id { }
   .c ~ span { }
   .d ~ [type] div { }
   ```

   Invalidation sets:

   ```
   .a { * }
   .b { #id }
   #id { }
   .c { * }
   .d { * }
   [type] { div }
   ```

Scheduling Invalidations

When we change a class attribute, id attribute, etc. we retrieve the descendant invalidation set for that property and add it to a list of scheduled invalidations for that element. The actual invalidation happens asynchronously and is scheduled to happen right before computed style recalculation.

The Style Invalidation and Style Recalculation

In the code, there are two passes over the DOM tree which are named Style Invalidation and Style Recalculation. Style Invalidation is the process of applying descendant invalidation sets scheduled for elements down the elements' subtree and mark elements for Style Recalculation. Style Recalculation then traverses the DOM tree and calculates the computed style (RenderStyle objects) for the marked elements.

An invalidation set is pushed onto the set of applied invalidation sets as an element for which the set is scheduled is entered during DOM tree traversal. The invalidation set is then being used to match its simple selector members against elements in the subtree, and pop it when we leave the subtree root that it was pushed for. Invalidation sets with the treeBoundaryCrossing flag set are propagated into shadow trees. Likewise, the insertionPointCrossing flag allows invalidation sets to apply across <content> elements into distributed nodes.

Example

HTML

```
<style>
.a .b { color: green }
#body #c { color: pink }
</style>
<body>
  <div class="b">
    <div id="t">
      <div class="b"></div>
      <div>
        <span id="c"></span>
      </div>
    </div>
  </div>
</body>
<script>
// force up-to-date style before applying changes
document.body.offsetTop;

document.body.id = "body";
t.classList.addClass("a");

// force recalc of changes
document.body.offsetTop;
</script>
```

Invalidation Sets

```
.a { .b }
.b { }
.c { }
#body { #c }
```

Description

In this example we schedule { #c } for the body element and { .b } for the #t element. The first set will match the span element with id=”c” and marked for recalc as it’s a descendant of body. The div child of body will not be marked for recalc as the second set is not pushed until we enter its child div with id=”t”, but the inner div with class=”b” will.

Notes and Further Improvements

Since we split simple selectors from rightmost compound selectors into distinct members of the invalidation sets, combinations of type selectors and classes/ids can be sub-optimal. Consider the selector “`.a span.b`”. Say you have a div with a large number of span descendants, only one of which has the class “b”, and set the class a on that div. Only the span with class “b” needs a style recalc, but since the invalidation set for “`.a`” becomes “`{ span, .b }`”, we will recalculate the computed style for all those span descendants.

Adjacent combinators are still very expensive. It is possible to extend the invalidation set concept to add Adjacent Invalidation Sets. There are several ways to implement that. One way is to do it indirectly through Descendant Invalidation Sets. An Adjacent Invalidation Set could contain the set of properties P found in the rightmost compound selector of an adjacent combinator chain. When such an invalidation set exist for a given change, schedule Descendant Invalidation Sets for the members of the Adjacent Invalidation Set on siblings of the modified element.

Example

Selectors

```
.a + .b {}
.c + .d + .e .f {}
.g ~ .h .i + .j {}
```

Descendant Invalidation Sets

```
.b { }
.e { .f }
```

```
.f { }
.h { .j }
.j { }
```

Adjacent Invalidation Sets

```
.a { .b }
.c { .e }
.d { .e }
.g { .h }
.i { .j }
```

As an example, when adding or removing the class “g” on an element, we could schedule an Adjacent Invalidation set for “`.g`” which in turn would schedule the Descendant Invalidation Set for “`.h`” on siblings matching “`.h`”.

Conclusion

Although Descendant Invalidation Sets will give you false positives when marking elements for style recalculation, it has proven to reduce the number of elements affected during style recalculation drastically, and hence reduced occurrences and severity of janks.