Moving DOM attributes to prototype chains [Part 5]

haraken@chromium.org

2013 Oct 30

Summary: We are planning to move DOM attributes to prototype chains and expose JavaScript getters/setters on DOM attributes. This change is necessary for correctness, cross-browser compatibility, and moving the Web forward. I propose to ship this change now even though it will regress performance of several micro benchmarks by 10% or so. We are planning to fix the regression in upcoming quarters.

What are we going to change?

Currently DOM attributes are defined on DOM objects. JavaScript getters/setters are not exposed on DOM attributes, like this:

```
div = document.createElement("div");
div.hasOwnProperty("id");  // true
div.__proto__.__proto__.hasOwnProperty("id");  // false
Object.getOwnPropertyDescriptor(div, "id");  // Object {value: "",
writable: true, enumerable: true, configurable: true}
Object.getOwnPropertyDescriptor(div.__proto__.__proto__, "id");  //
undefined
```

However, the current behavior is wrong per the Web IDL spec. Per the spec, we should move DOM attributes from DOM objects to JavaScript prototype chains and expose JavaScript getters/setters on DOM attributes. Specifically, we are planning to change the behavior as follows:

```
div = document.createElement("div");
div.hasOwnProperty("id");  // false
```

```
div.__proto__.__proto__.hasOwnProperty("id");  // true
Object.getOwnPropertyDescriptor(div, "id");  // undefined
Object.getOwnPropertyDescriptor(div.__proto__.__proto__, "id");  //
Object {get: function, set: function, enumerable: false, configurable:
false}
```

What is the benefit?

There are a lot of benefits to making the change:

- The new behavior is conformant with the Web IDL spec.

- The new behavior is already implemented in IE and Firefox.

- The new behavior enables developers to hook DOM attribute getters/setters. This will improve hackability of DOM operations. It will enable developers to implement Polyfill and JavaScript libraries that override default DOM attribute behaviors. In particular, Polyfill implementation will become much easier and faster.

- The new behavior makes it easy to port Blink's C++ implementation into JavaScript and reduce complexity of Blink. For example, we are planning to move editing implementation (e.g., execCommand) from C++ to JavaScript. With the new behavior, we just need to hook DOM attribute getters/setters and implement the logic all in JavaScript.

- The new behavior is necessary for custom elements in the Web Components, which we are planning to ship in M33. In custom elements, developers can define customized DOM elements that inherit from existing DOM elements. Here developers want to override DOM attribute behavior of existing DOM elements by hooking their JavaScript getters/setters.

In summary, this change is necessary for correctness, cross-browser compatibility, and moving the Web forward.

What about compatibility risk?

The compatibility risk is low, because IE and Firefox already implemented the new behavior more than one year ago. On the other hand, Safari has not yet implemented the new behavior (See this WebKit bug).

What about performance?

Summary

This change regresses performance of micro benchmarks by 8.7%. On the other hand, we observe no regression in real-world benchmarks. We have been working on the performance optimization for two years, and are concluding that it would be hard to kill the 8.7% regression without implementing a complicated optimization which will take three quarters. Given that it wouldn't make sense to defer shipping the change and continue bothering developers just for a couple of regressions observed in micro benchmarks, I propose to ship the change now even though it will slightly regress performance of micro benchmarks. We are planning to fix the regression in upcoming quarters after shipping.

Details

When I started working on this project two years ago, we observed 70% regression in Dromaeo. Thanks to great efforts of the V8 team and binding team, now we observe only 8.7% regression. Below is the result of Dromaeo benchmarks in 64-bit Linux Chrome:

```
[dom-attr]     1078 runs/sec =>  1080 runs/sec (+0.2%)
[dom-modify]    459 runs/sec =>   468 runs/sec (+1.9%)
[dom-query]   23959 runs/sec => 23050 runs/sec (-3.9%)
[dom-traverse]  573 runs/sec =>   527 runs/sec (-8.7%)
```

This result looks reasonable for the following reasons:

- In theory, the performance regression comes from the fact that in the new behavior we have to look up a prototype chain to find a given DOM attribute. In other words, the performance regression exists in the pure overhead to callback C++ code in Blink from JIT code in V8.

- [dom-traverse] is just traversing DOM trees using .firstChild, .parentNode etc. The Blink implementation of .firstChild, .parentNode etc is perfectly optimized and they do almost nothing in Blink. Thus, the performance of [dom-traverse] is dominated by the pure overhead of binding callbacks. That's why we observe the 8.7% regression in [dom-traverse].

Also, we observe worse regressions in more pathological micro benchmarks in PerformanceTests/Bindings/. For example, in PerformanceTests/Bindings/first-child.html (which just repeats div.firstChild), we observe 33.4% regression.

```
[first-child.html] 954 runs/sec => 715 runs/sec (-33.4%)
```

Furthermore, I can expect a bit larger regressions in Windows, since the optimization of Visual Studio tends to be weaker than clang++ or g++. (The above result is a result in 64-bit Linux machine.)

HOWEVER, all of these are just results of micro benchmarks. We cannot observe any regression in Robohornet.

Justification for the performance regression

Chrome and Blink follow a no-regression policy. Thus, in general we are not allowed to regress performance at all even in micro benchmarks. In particular case of DOM binding area, we have allowed small regressions in pathological micro benchmarks in PerformanceTests/Bindings, but we have not allowed more than 2% regressions in Dromaeo. Given the situation, I would like to discuss with you if the 8.7% regression in Dromaeo is acceptable or not for making this specific change.

As mentioned above, we started working on the project two years ago and have decreased the regression from 70% to 8.7%. The remaining overhead comes from the pure overhead of binding callbacks. We have spent a lot of time to decrease the regression and are concluding that it would be hard to decrease the regression any more without “crankshafting binding callbacks to V8”. (If you are interested in the history, you can look at these previous documents: 1, 2, 3, 4.)

“Crankshafting” means the following optimization. Imagine the following code:

```
for (var i = 0; i < 100000; i++)
  div.firstChild;
```

Currently, V8 has to set up a lot of things every time V8 calls back a C++ callback for div.firstChild. For example, V8 has to set up PropertyCallbackInfo struct, create a HandleScope, check that div has a correct type etc. However, it is inefficient to do the set up every time. Instead, V8 should do the set up only once before starting the for loop. Then the div.firstChild access will become a simple jump to a C++ callback, which will definitely be faster. This optimization becomes possible if we move the binding callback logic into Crankshaft (the JIT compiler of V8). This optimization will not only resolve our regression problem but also improve all binding performance in general. I am expecting significant speedup in real-world DOM bindings. We are pretty sure that this is a right way we should move on in long term.

However, the problem is that it will take about three quarters to implement the optimization. Thus the question here is whether we should wait for the optimization and then make our change, or we should make our change and then do the optimization. If we should stick to the no regression policy, we should wait for the optimization. However, in this specific case, I think the advantage of making this change now would be larger than the advantage of sticking to the

no regression policy for micro benchmarks. Thus I would propose to make the change even though it will regress performance of micro benchmarks.

I would like to hear your thoughts.

What is a shipping plan?

I am planning the following steps:

> (1) Move popular 20 DOM attributes (used in Dromaeo) to prototype chains and expose JavaScript getters/setters.
>
> (2) Watch perf bots. If we observe unacceptable regressions in some perf bots, we will change our plan to work on the Crankshaft optimization first.
>
> (3) Verify that the change doesn't cause compatibility issues in real worlds.
>
> (4) Move all DOM attributes to prototype chains and expose JavaScript getters/setters.

I hope I can finish everything by the end of Q4.