Fast frozen & sealed elements in V8

Attention: Shared Google-externally

Authors: bmeurer@, leszeks@, verwaest@, jarin@, duongn@microsoft.com

Last Updated: 2019-04-09

TL;DR This document describes the issue of `Object.freeze()`, `Object.seal()`, and `Object.preventExtensions()` turning arrays into *dictionary mode*, and thus making them slower to access. It also explores two potential solutions to the problem.

Short Link bit.ly/fast-frozen-sealed-elements-in-v8

Bug v8:6831

Background

The integrity level control builtins `Object.freeze()`, `Object.seal()`, and `Object.preventExtensions()` were introduced with the ECMAScript 5.1 specification, and they are being used quite a bit already. Later, the ECMAScript 2015 specification introduced so-called tagged templates, which implicitly use `Object.freeze()`. More recently, TC39 delegates kicked off a proposal for some yet-to-be-defined way to freeze object prototypes.

Currently, the use of these builtins incurs a performance cost in V8, since they turn arrays into `DICTIONARY_ELEMENTS` mode, making any accesses on them slower.

Tagged templates

Tagged templates call `Object.freeze()` on the `strings` and the `strings.raw` arrays to make sure that user code does not mess with their values. These `strings` and the `strings.raw` arrays are available in the tag function, i.e.:

```
function tag(strings, ...values) {
  let s = strings[0];
  for (let i = 1; i < strings.length; ++i) {
    s += values[i - 1] + strings[i];
  }
  return s;
}

tag`Hello ${41 + 1}!\n`;
```

```
// → "Hello 42!\n"
```

The `strings.raw` provides access to the *raw string literals*, i.e. in the example `strings[1]` contains `"\n"` whereas `strings.raw[1]` contains `"\\n"`.

Tagged templates are becoming increasingly popular, with more and more libraries and frameworks adopting it. For example, the following popular libraries make heavy use of the tagged templates feature:

1. lit-html & LitElement (Polymer)
2. styled-components
3. htm
4. common-tags

For example lit-html lets you write HTML templates in JavaScript, and then efficiently renders (and re-renders) those templates together with data to create (and update) DOM, i.e.:

```
import {html, render} from 'lit-html';

// A lit-html template uses the `html` template tag:
let sayHello = (name) => html`<h1>Hello ${name}</h1>`;

// It's rendered with the `render()` function:
render(sayHello('World'), document.body);

// And re-renders only update the data that changed, without
// VDOM diffing!
render(sayHello('Everyone'), document.body);
```

This in turn is used by LitElement to provide lightweight web components utilising the shadow DOM. In styled-components tagged templates are used to provide easy inline styling for React components, like in this example:

```
const Button = styled.a`
  display: inline-block;
  border-radius: 3px;
  padding: 0.5rem 0;
  margin: 0.5rem 1rem;
  width: 11rem;
  background: transparent;
  color: white;
  border: 2px solid white;
  ${props => props.primary && css`
    background: white;
    color: palevioletred;
  `}
```

```
`;
```

Performance work-around

There's a (sort of) known work-around for the performance problem, which is to use Babel in the so-called *loose mode*, which skips the freezing of the strings and strings.raw arrays (unlike regular Babel transpilation which does the freezing properly). For example

```
tag`Hello ${42}!\n`
```

normally translates to

```
var _templateObject = _taggedTemplateLiteral(
  ["Hello ", "!\n"],
  ["Hello ", "!\\n"]
);
function _taggedTemplateLiteral(strings, raw) {
  return Object.freeze(
    Object.defineProperties(strings, { raw: { value: Object.freeze(raw) } })
  );
}
tag(_templateObject, 42)
```

when using the `es2015` preset. So both the strings and the strings.raw array are properly frozen using `Object.freeze()`. However in *loose mode* Babel will generate a simpler version of the `_taggedTemplateLiteral` function above instead, which looks like this

```
function _taggedTemplateLiteralLoose(strings, raw) {
  strings.raw = raw;
  return strings;
}
```

and thus doesn't do any freezing of either the `strings` or the `strings.raw` array (also the raw property is initialized using a property assignment instead of `Object.defineProperties()`, which means that it's going to be configurable, enumerable and writable in *loose mode*, but that's probably a less relevant detail).

TypeScript

For reference, TypeScript by default generates the following code for the above mentioned example

```
var __makeTemplateObject = (this && this.__makeTemplateObject) || function
(cooked, raw) {
```

```
    if (Object.defineProperty) { Object.defineProperty(cooked, "raw", {
value: raw }); } else { cooked.raw = raw; }
    return cooked;
};
tag(__makeTemplateObject(["Hello ", "!\n"], ["Hello ", "!\\n"]), 42);
```

so it behaves like Babel by default when targeting ES3/ES5.

Freezing API objects

`Object.freeze()`, `Object.seal()`, and `Object.preventExtensions()` are often used to express that certain objects are not supposed to be extended and/or changed. This helps when designing / implementing safe APIs. The most popular alternative to freezing objects exposed by APIs is doing *defensive copies*, which comes at a cost though, and might not match the intended semantics. Unfortunately using any of the above mentioned builtins on arrays currently turns these arrays into *dictionary mode* in the V8 engine, and thus makes it slower to operate on these arrays.

Currently most frameworks limit the use of above mentioned integrity level control builtins to debug builds, mostly because of the performance penalty.

FrozenArray in WebIDL

A special case of frozen API object is the FrozenArray array type in WebIDL, which is already used a bunch in Blink.

Frozen objects and shape transitions (orthogonal)

There's an orthogonal issue in V8, which is also related to the use of `Object.freeze()`, `Object.seal()`, and `Object.preventExtensions()`, and which affects regular objects as well. Here certain internal shape transitions in V8 don't play well together with any of the above, which can lead to drastic performance cliffs (i.e. in one case it was observed that the performance dropped by **more than a 100x**). This is tracked by v8:8538 and there's a document Improve handling of Object.(seal|preventExtensions) that describes the current proposed solution.

Performance impact

There are two interesting scenarios to consider: the **latency** case and the **throughput** case.

The first case (**latency**) is mostly relevant when using something like lit-html or styled-components in the context of the browser to perform client-side rendering. In this case

V8 is probably mostly executing code in the interpreter, hasn't reached a steady state, and thus the impact of having dictionary elements for the `strings` and `strings.raw` arrays is probably less dominant, while still important (it's not unlikely that a simple Polymer application has to render hundreds of templates quickly during startup).

The second case (**throughput**) becomes relevant when using for example tagged templates in a server-side rendering scenario, i.e. in a template engine running on top of express, or in a potential version of lit-html that does server-side rendering to speed up the first page load[1]. Focusing on this case is becoming more and more critical as popular sites move towards server-side rendering for the initial page load.

I created a small benchmark to measure the impact of the `strings` array going to dictionary mode:

```
function tag(strings, ...values) {
  let a = 0;
  for (let i = 0; i < strings.length; ++i) a += strings[i].length;
  return a;
}

function driver(n) {
  let result = 0;
  for (let i = 0; i < n; ++i) {
    result += tag`${"Hello"} ${"cruel"} ${"slow"} ${"world"}!\n`;
    result += tag`${"Why"} ${"is"} ${"this"} ${"so"} ${"damn"}
${"slow"}?!\n`;
  }
  return result;
}

driver(10);
driver(100);
driver(1000);
driver(10000);

console.time('Time');
driver(1e6);
console.timeEnd('Time');
```

It does some warm-up in the beginning to make sure we focus the measurement on the difference in the `strings` elements, and not on other factors that are not relevant in the context of this document. We compare the code above to a version produced by Babel in *loose mode*:

```
var _templateObject = _taggedTemplateLiteralLoose(
    ["", " ", " ", " ", "!\n"],
```

[1] There's already a package called lit-html-server that provides similar functionality to lit-html on the server side.

```
    ["", " ", " ", " ", "!\\n"]
  ),
  _templateObject2 = _taggedTemplateLiteralLoose(
    ["", " ", " ", " ", " ", "?!\n"],
    ["", " ", " ", " ", " ", "?!\\n"]
  );

function _taggedTemplateLiteralLoose(strings, raw) {
  strings.raw = raw;
  return strings;
}

function tag(strings) {
  var a = 0;
  for (var i = 0; i < strings.length; ++i) {
    a += strings[i].length;
  }
  return a;
}

function driver(n) {
  var result = 0;
  for (var i = 0; i < n; ++i) {
    result += tag(_templateObject, "Hello", "cruel", "slow", "world");
    result += tag(_templateObject2, "Why", "is", "this", "so", "damn",
"slow");
  }
  return result;
}

driver(10);
driver(100);
driver(1000);
driver(10000);

console.time("Time");
driver(1e6);
console.timeEnd("Time");
```

We run this test with latest V8, once in the default configuration emulating the **throughput** case, and another time passing `--noopt` to completely disable the optimizing compiler and thereby emulating the **latency** case (we take the average of 10 individual runs):

	default (**throughput**)	`--noopt` (**latency**)
Original	187 ms	767 ms
Babel loose mode	21 ms	640 ms

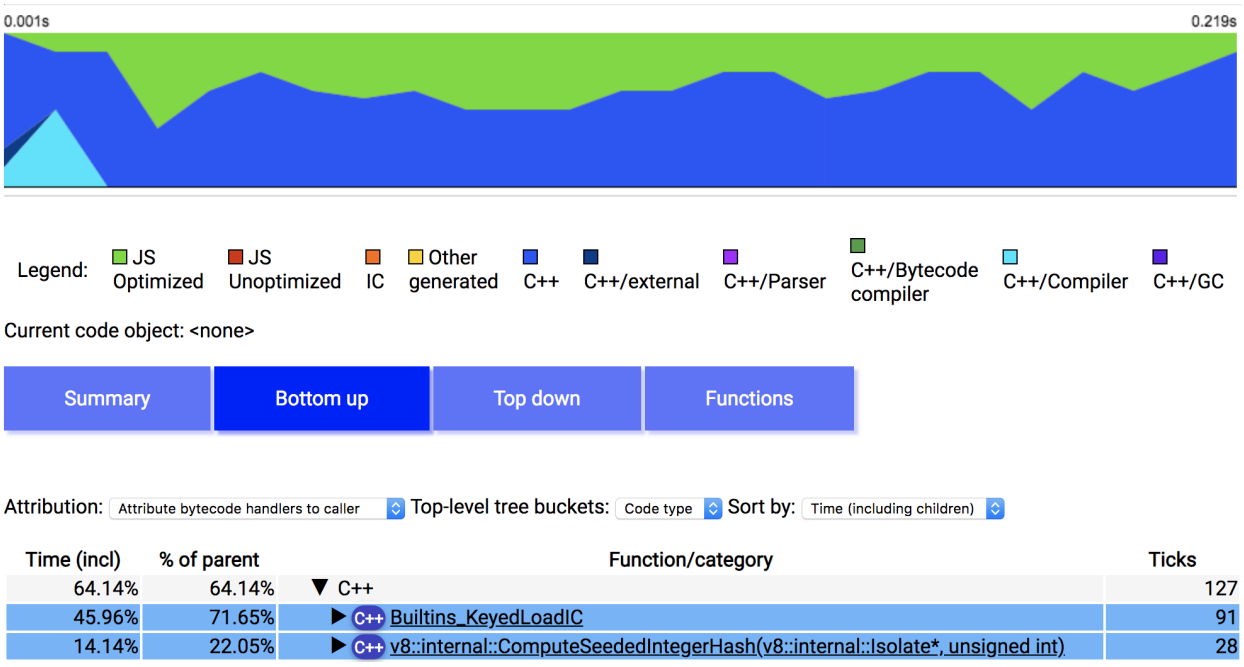
As you can see, the Babel transpiled version wins in both cases. But what's interesting to note is

that the effect on the peak performance case is a lot more severe with a **slowdown of 9x** versus a mere **20% slowdown** in the latency case. So server-side rendering using tagged templates is affected a lot more than when using tagged templates for initial rendering on the client-side.

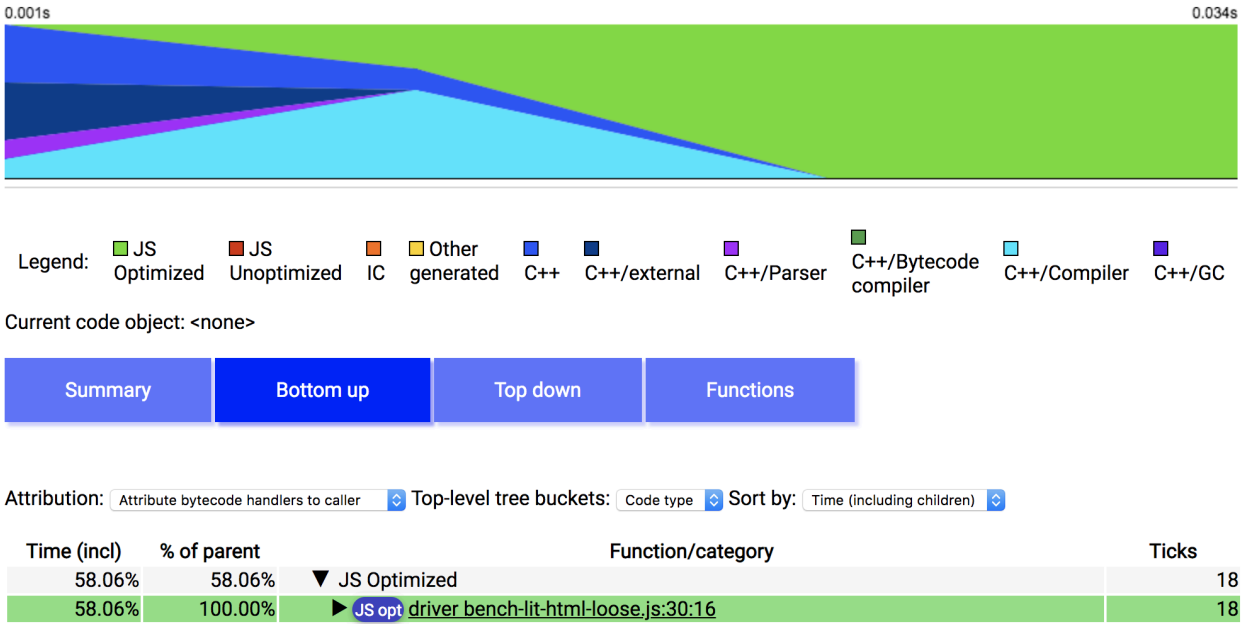
Note: Of course using tagged templates for rendering is just one possible use case. There are plenty of others that might be equally relevant, but this document is mostly focusing on this case for now.

Profiling peak performance

If we profile the peak performance case using `--prof` we can clearly see that in the original ES2015 code V8 spends significant amount of time in the KeyedLoadIC builtin, which handles the case of keyed property access to dictionary arrays:



Whereas in the Babel loose mode code, where the test terminates in 34ms versus 219ms above, everything can be handled fast in optimized code:



Proposed solutions

There are two possible solutions that I can think of which would make sense at this point. We could either introduce new elements kinds to express the frozen/sealed state, and have the various places in the code that deal with extending elements backing stores check the extensible bit, or we could teach TurboFan about dictionary elements (actually both are somewhat orthogonal and if necessary we could do both).

	Pros	Cons
TurboFan dictionary elements	• Easier to implement (local change in TurboFan)	• Only recovers small portion of the performance difference • Even more places that need to worry about the hashing of integer keys
Special elements kinds	• Maximum peak performance • Potential for even more optimization (utilising the immutability of the elements) • Easier to provide	• More work to implement initially • Adds more elements kinds

	consistent performance (i.e. fast array iteration builtins)	

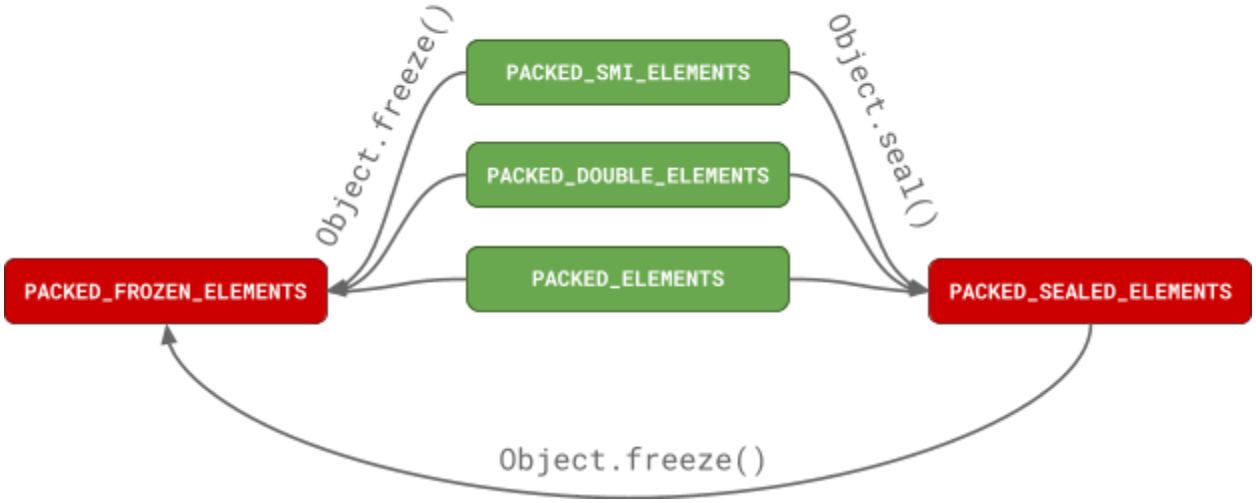
Implement special elements kinds (preferred solution)

As described in Elements kinds in V8, the V8 engine uses a concept called *elements kinds* to optimize array backing stores. There are various *fast elements kinds* and there's the so-called *dictionary elements* (called `DICTIONARY_ELEMENTS` inside of V8). Currently the *fast elements kinds* all imply that the individual array indexed properties are all *configurable*, *enumerable*, and *writable*, and on top of that, new elements can be added to the array (i.e. `Object.preventExtensions()` wasn't called on that array).

Now we could add new elements kinds

- `PACKED_SEALED_ELEMENTS`,
- `PACKED_FROZEN_ELEMENTS`,
- `HOLEY_SEALED_ELEMENTS`, and
- `HOLEY_FROZEN_ELEMENTS`,

which would correspond to `PACKED_ELEMENTS` and `HOLEY_ELEMENTS` respectively, but with the additional constraint that all elements are *non-configurable* (in case of `SEALED`) or also *non-writable* (in case of `FROZEN`). In theory we could even add fine grained `PACKED_SEALED_SMI_ELEMENTS` and `PACKED_SEALED_DOUBLE_ELEMENTS` (and same for frozen and holey combinations), but that is probably too much complexity for the purpose of what we're trying to accomplish here.



We end up with the above transitions (and similar for the HOLEY_ELEMENTS). The interesting bit here is that we might be able to leverage the fact that once the backing store is in PACKED_FROZEN_ELEMENTS state, we know that there won't be any more elements transitions, i.e. the backing store is not going to change. The same for PACKED_SEALED_ELEMENTS, where it can only transition to PACKED_FROZEN_ELEMENTS state, and that wouldn't change the backing store. In both cases we could also turn the backing store itself into *copy-on-write* mode and easily share it and bake it directly into optimized code.

On top of that all builtins and code-paths that add new array elements would have to check the `is_extensible` bit on the object shape to respect `Object.preventExtensions()` even if the elements are in *fast mode*, before they can add more elements to the backing store.

Implementation considerations

CLs 1461166, 1474559, 1481895

We need to add performance tests to verify the improvements. A first set of tests was added in 1461166. We might need to add more later.

Then concrete action items are probably along the lines of:

1. We need a lot more correctness tests for this, since the current coverage doesn't even catch basic mistakes.
    a. Add more tests for non-extensible, sealed, frozen packed-elements object in 1481895, 1544274, 1531030
2. We might want to start with just treating PACKED_SEALED_ELEMENTS and PACKED_FROZEN_ELEMENTS (and their HOLEY counterparts) like DICTIONARY_ELEMENTS in CodeStubAssembler and TurboFan.
    a. Just take the slow-path for them, potentially going to runtime. And then later gradually allow fast-paths for the interesting cases.
3. We will likely need both HOLEY and PACKED support for frozen/sealed, otherwise this will be a weird performance cliff.

Clever bit field encoding for the elements kinds

TBD

Teach TurboFan about dictionary elements (alternative solution)

Currently TurboFan and most of the Array builtins don't support DICTIONARY_ELEMENTS. Instead those usually go through the generic code paths (i.e. the KeyedLoadIC), and involve a bit of C++ code (to compute the hash for the integer key). We could inline some of this logic into TurboFan for loading the value from the backing store once the hash code is computed, that

would allow us to at least perform some optimizations across these elements accesses. It'd still be quite a bit slower than *fast mode* arrays.