V8 JavaScript Engine

Wednesday, August 30, 2017

Fast Properties in V8

In this blog post we would like to explain how V8 handles JavaScript properties internally. From a JavaScript point of view there are only a few distinctions necessary for properties. JavaScript objects mostly behave like dictionaries, with string keys and arbitrary objects as values. The specification does however treat integer-indexed properties and other properties differently during iteration. Other than that, the different properties behave mostly the same, independent of whether they are integer indexed or not.

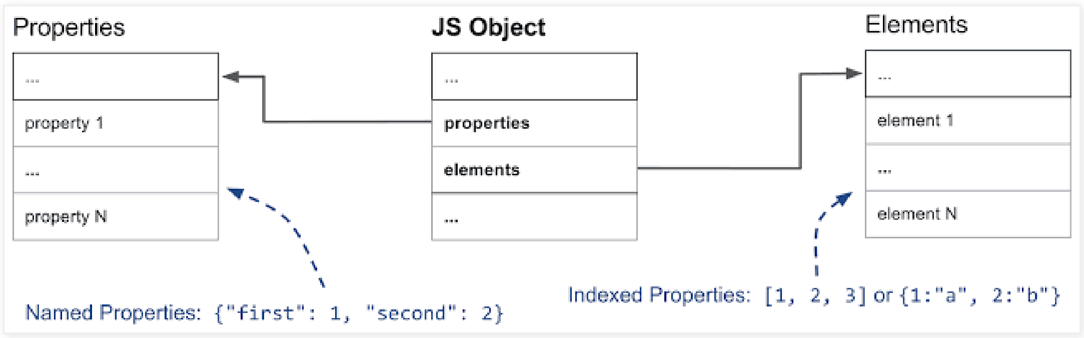
However, under the hood V8 does rely on several different representations of properties for performance and memory reasons. In this blog post we are going to explain how V8 can provide fast property access while handling dynamically-added properties. Understanding how properties work is essential for explaining how optimizations such as inline caches work in V8.

This post explains the difference in handling integer-indexed and named properties. After that we show how V8 maintains HiddenClasses when adding named properties in order to provide a fast way to identify the shape of an object. We'll then continue giving insights into how named properties are optimized for fast accesses or fast modification depending on the usage. In the final section we provide details on how V8 handles integer-indexed properties or array indices.

Named Properties vs. Elements

Let's start by analysing a very simple object such as `{a: "foo", b: "bar"}`. This object has two named properties, "a" and "b". It does not have any integer indices for property names. array-indexed properties, more commonly known as elements, are most prominent on arrays. For instance the array `["foo", "bar"]` has two array-indexed properties: `0`, with the value `"foo"`, and `1`, with the value `"bar"`. This is the first major distinction on how V8 handles properties in general.

The following diagram shows what a basic JavaScript object looks like in memory.



Elements and properties are stored in two separate data structures which makes adding and accessing properties or elements more efficient for different usage patterns.

Elements are mainly used for the various Array.prototype methods such as pop or slice. Given that these functions access properties in consecutive ranges, V8 also represents them as simple arrays internally — most of the time. Later in this post we will explain how we sometimes switch to a sparse dictionary-based representation to save memory.

Named properties are stored in a similar way in a separate array. However, unlike elements, we cannot simply use the key to deduce their position within the properties array; we need some additional metadata. In V8 every JavaScript object has a HiddenClass associated. The HiddenClass stores information about the shape of an object, and among other things, a mapping from property names to indices into the properties. To complicate things we sometimes use a dictionary for the properties instead of a simple array. We will explain this in more detail in a dedicated section.

Takeaway from this section:

About

This is the official blog of the V8 JavaScript engine.

More information, docs, and tutorials about V8: https://github.com/v8/v8/wiki

V8 issue tracker: https://bugs.chromium.org/p/v8/issues/list

Updates from the Chromium project: http://blog.chromium.org

Search This Blog

Search

Blog Archive

▼ 2017 (16)
- ▼ August (3)
  - Fast Properties in V8
  - About that hash flooding vulnerability in Node.js......
  - V8 Release 6.1
- ► July (1)
- ► June (1)
- ► May (2)
- ► April (2)
- ► March (2)
- ► February (4)
- ► January (1)

► 2016 (19)
► 2015 (11)

Twitter

Tweets by @v8js

V8 Retweeted

Franziska Hinkelm…
@fhinkel

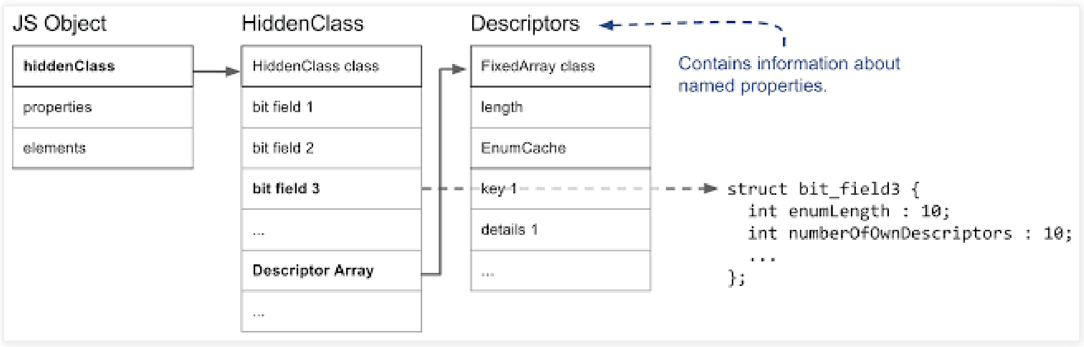
Look, runtime type information (string, number, null, custom classes, ...) for JavaScript 😍
github.com/fhinkel/type-p…

- Array-indexed properties are stored in a separate elements store.
- Named properties are stored in the properties store.
- Elements and properties can either be arrays or dictionaries.
- Each JavaScript object has a HiddenClass associated that keeps information about the object shape.

HiddenClasses and DescriptorArrays

After explaining the general distinction of elements and named properties we need to have a look at how HiddenClasses work in V8. This HiddenClass stores meta information about an object, including the number of properties on the object and a reference to the object's prototype. HiddenClasses are conceptually similar to classes in typical object-oriented programming languages. However, in a prototype-based language such as JavaScript it is generally not possible to know classes upfront. Hence, in this case V8, HiddenClasses are created on the fly and updated dynamically as objects change. HiddenClasses serve as an identifier for the shape of an object and as such a very important ingredient for V8's optimizing compiler and inline caches. The optimizing compiler for instance can directly inline property accesses if it can ensure a compatible objects structure through the HiddenClass.

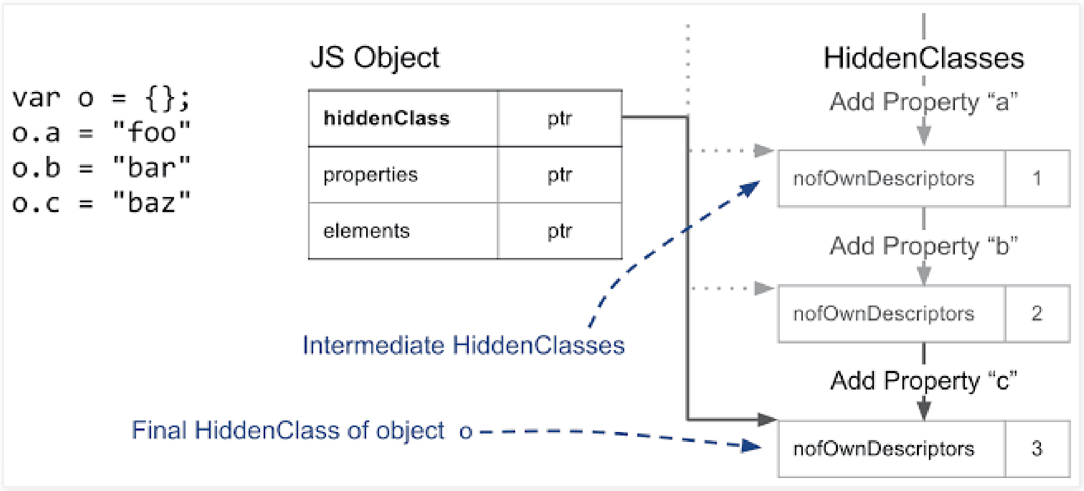
Let's have a look at the important parts of a HiddenClass.



```
struct bit_field3 {
  int enumLength : 10;
  int numberOfOwnDescriptors : 10;
  ...
};
```

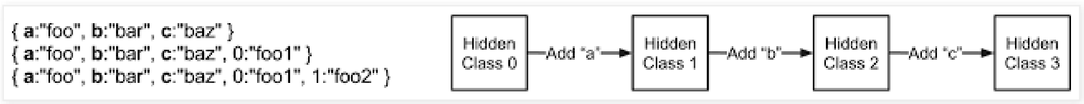
In V8 the first field of a JavaScript object points to a HiddenClass. (In fact, this is the case for any object that is on the V8 heap and managed by the garbage collector.) In terms of properties, the most important information is the third bit field, which stores the number of properties, and a pointer to the descriptor array. The descriptor array contains information about named properties like the name itself and the position where the value is stored. Note that we do not keep track of integer indexed properties here, hence there is no entry in the descriptor array.

The basic assumption about HiddenClasses is that objects with the same structure — e.g. the same named properties in the same order — share the same HiddenClass. To achieve that we use a different HiddenClass when a property gets added to an object. In the following example we start from an empty object and add three named properties.



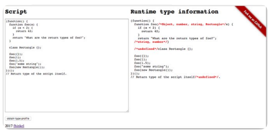
```
var o = {};
o.a = "foo"
o.b = "bar"
o.c = "baz"
```

Every time a new property is added, the object's HiddenClass is changed. In the background V8 creates a transition tree that links the HiddenClasses together. V8 knows which HiddenClass to take when you add, for instance, the property "a" to an empty object. This transition tree makes sure you end up with the same final HiddenClass if you add the same properties in the same order. The following example shows that we would follow the same transition tree even if we add simple indexed properties in between.



```
{ a:"foo", b:"bar", c:"baz" }
{ a:"foo", b:"bar", c:"baz", 0:"foo1" }
{ a:"foo", b:"bar", c:"baz", 0:"foo1", 1:"foo2" }
```

However, if we create a new object that gets a different property added, in this case property "d," V8 creates a separate branch for the new HiddenClasses.



7h

V8 Retweeted

Sathya Gunasekar…
@_gsathya

Intent to ship: Promise.prototype.finally groups.google.com/forum/#!topic/…

Intent to …
Posted 9/…
groups.g…

Sep 8, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

❓ #JavaScript poll: Is `class A extends Object {...}` a thing that you'd use (and if yes, why)? Or do you stick to `class A {...}` instead?

Sep 8, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

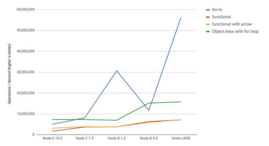
I'm highly interested to hear your @NodeJS 8.3.0 performance stories! Any serious regressions? Any welcome improvements? 📣 🔥 #javascript

Aug 14, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

💉 Just published "Restoring for..in peak performance" with some background on the latest for..in goodness 🔥 benediktmeurer.de/2017/09/07/res… #V8

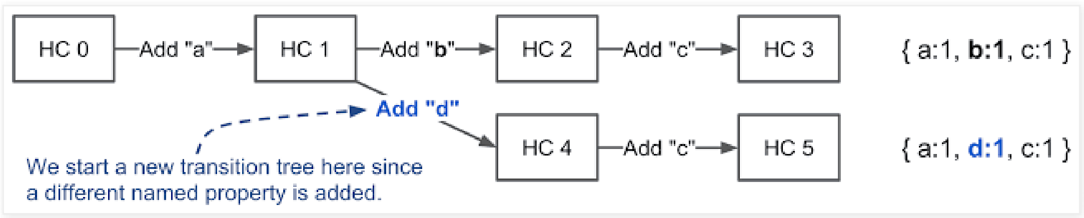


Sep 7, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

📣 This might not look like a lot, but saving a word per closure translates to 500kB savings on several web pages. 🔥 chromium-



Takeway from this section:

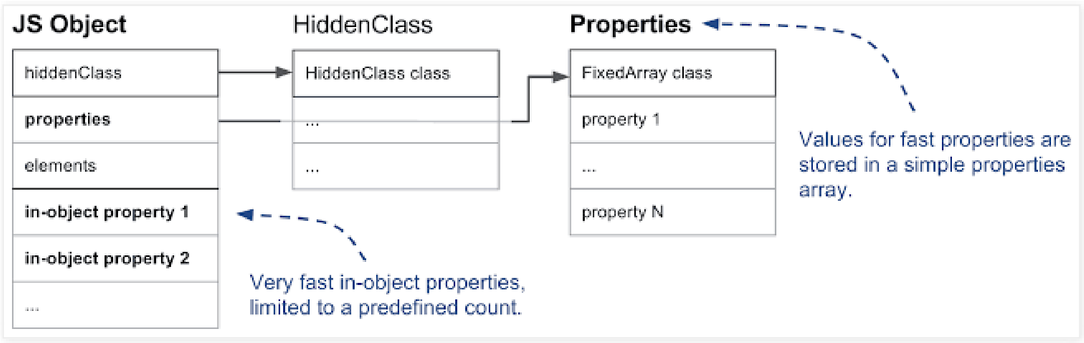
- Objects with the same structure (same properties in the same order) have the same HiddenClass
- By default every new named property added causes a new HiddenClass to be created.
- Adding array-indexed properties does not create new HiddenClasses.

The Three Different Kinds of Named Properties

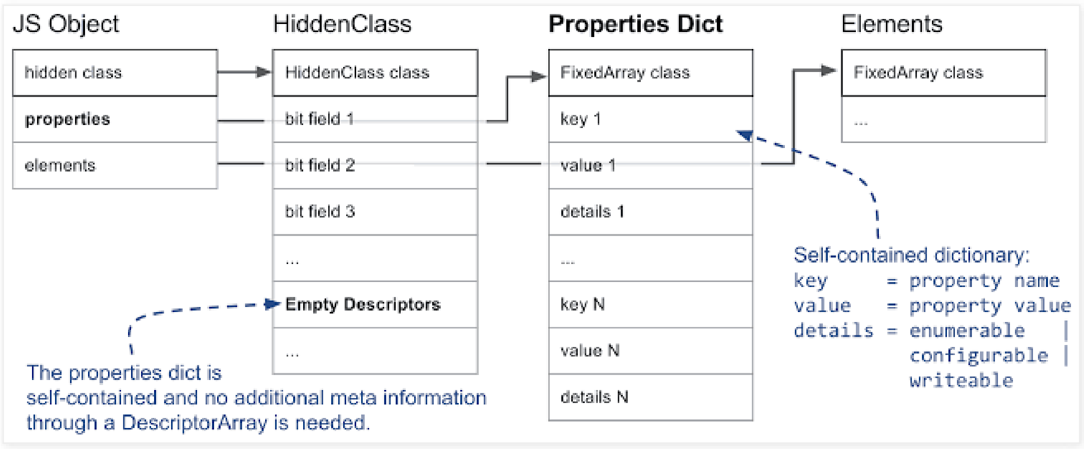
After giving an overview on how V8 uses HiddenClasses to track the shape of objects let’s dive into how these properties are actually stored. As explained in the introduction above, there are two fundamental kind of properties: named and indexed. The following section covers named properties.

A simple object such as `{a: 1, b: 2}` can have various internal representations in V8. While JavaScript objects behave more or less like simple dictionaries from the outside, V8 tries to avoid dictionaries because they hamper certain optimizations such as inline caches which we will explain in a separate post.

In-object vs. Normal Properties: V8 supports so-called in-object properties which are stored directly on the object themselves. These are the fastest properties available in V8 as they are accessible without any indirection. The number of in-object properties is predetermined by the initial size of the object. If more properties get added than there is space in the object, they are stored in the properties store. The properties store adds one level of indirection but can be grown independently.



Fast vs. Slow Properties: The next important distinction is between fast and slow properties. Typically we define the properties stored in the linear properties store as "fast". Fast properties are simply accessed by index in the properties store. To get from the name of the property to the actual position in the properties store, we have to consult the descriptor array on the HiddenClass, as we've outlined before.

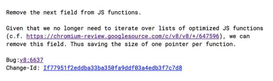


However, if many properties get added and deleted from an object, it can generate a lot of time and memory overhead to maintain the descriptor array and HiddenClasses. Hence, V8 also supports so-called slow properties. An object with slow properties has a self-contained dictionary as a properties store. All the properties meta information is no longer stored in the descriptor array on the HiddenClass but directly in the properties dictionary. Hence, properties can be added and removed without updating the HiddenClass. Since inline caches don’t work with dictionary properties, the latter, are typically slower than fast properties.

Takeaway from this section:

- There are three different named property types: in-object, fast and slow/dictionary.

review.googlesource.com/c/v8/v8/+/6523…

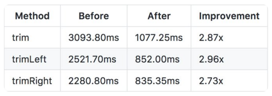

```
Remove the next field from JS functions.

Given that we no longer need to iterate over lists of optimized JS functions
(c.f. https://chromium-review.googlesource.com/c/v8/v8/+/647596), we can
remove this field. Thus saving the size of one pointer per function.

Bug:v8:6637
Change-Id: If77951f2eddba33ba350fa9ddf03a4edb3f7c7d8
```

Sep 7, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

Awesome to see @peterwmwong boosting String builtins in @v8js (guided by @schuay). Today String#trim got a boost. 🔥 chromium-review.googlesource.com/c/v8/v8/+/6476…



Method	Before	After	Improvement
trim	3093.80ms	1077.25ms	2.87x
trimLeft	2521.70ms	852.00ms	2.96x
trimRight	2280.80ms	835.35ms	2.73x

Sep 6, 2017

V8 Retweeted

Kunal
@KuXMLP

Thanks for the insights. Similar improvement opportunity exist for Chakracore as well. github.com/Microsoft/Chak…

Object c…
There wa…
github.com

Sep 3, 2017

V8 Retweeted

小倉唯
@Constellation

JavaScriptCore also has this :) github.com/WebKit/webkit/… #WebKit

WebKit/…
Unofficial…
github.com

Sep 2, 2017

V8 Retweeted

SpiderMonkey
@SpiderMonkeyJS

SpiderMonkey now also optimizes the Object(object) pattern used in webpack generated code. bugzil.la/1395513

Sep 1, 2017

V8
@v8js

Replying to @v8js

webpack v3 generates code

1. In-object properties are stored directly on the object itself and provide the fastest access.
2. Fast properties live in the properties store, all the meta information is stored in the descriptor array on the HiddenClass.
3. Slow properties live in a self-contained properties dictionary, meta information is no longer shared through the HiddenClass.

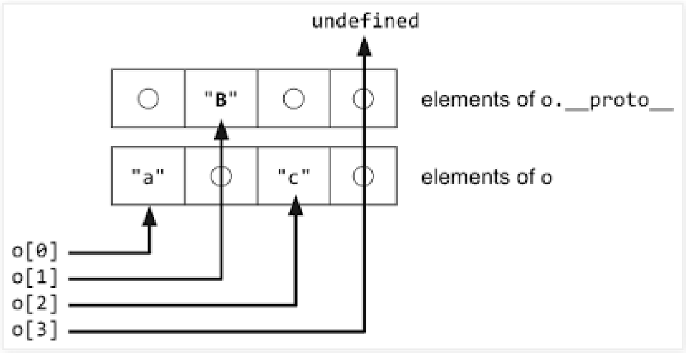
- Slow properties allow for efficient property removal and addition but are slower to access than the other two types.

Elements or array-indexed Properties

So far we have looked at named properties and ignored integer indexed properties commonly used with arrays. Handling of integer indexed properties is no less complex than named properties. Even though all indexed properties are always kept separately in the elements store, there are 20 different types of elements!

Packed or Holey Elements: The first major distinction V8 makes is whether the elements backing store is packed or has holes in it. You get holes in a backing store if you delete an indexed element, or for instance, you don't define it. A simple example is `[1,,3]` where the second entry is a hole. The following example illustrates this issue:

```
const o = ["a", "b", "c"];
console.log(o[1]);          // Prints "b".

delete o[1];                // Introduces a hole in the elements store.
console.log(o[1]);          // Prints "undefined"; property 1 does not exist.
o.__proto__ = {1: "B"};     // Define property 1 on the prototype.

console.log(o[0]);          // Prints "a".
console.log(o[1]);          // Prints "B".
console.log(o[2]);          // Prints "c".
console.log(o[3]);          // Prints undefined
```



In short, if a property is not present on the receiver we have to keep on looking on the prototype chain. Given that elements are self-contained, e.g. we don't store information about present indexed properties on the HiddenClass, we need a special value, called the_hole, to mark properties that are not present. This is crucial for the performance of Array functions. If we know that there are no holes, i.e. the elements store is packed, we can perform local operations without expensive lookups on the prototype chain.

Fast or Dictionary Elements: The second major distinction made on elements is whether they are fast or dictionary-mode. Fast elements are simple VM-internal arrays where the property index maps to the index in the elements store. However, this simple representation is rather wasteful for very large sparse/holey arrays where only few entries are occupied. In this case we used a dictionary-based representation to save memory at the cost of slightly slower access:

```
const sparseArray = [];
sparseArray[9999] = "foo"; // Creates an array with dictionary elements.
```

In this example, allocating a full array with 10k entries would be rather wasteful. What happens instead is that V8 creates a dictionary where we store a key-value-descriptor triplets. The key in this case would be 9999 and the value `"foo"` and the default descriptor is used. Given that we don't have a way to store descriptor details on the HiddenClass, V8 resorts to slow elements whenever you define an indexed properties with a custom descriptor:

```
const array = [];
Object.defineProperty(array, 0, {value: "fixed", configurable: false});
console.log(array[0]);      // Prints "fixed".
array[0] = "other value";   // Cannot override index 0.
console.log(array[0]);      // Still prints "fixed".
```

In this example we added a non-configurable property on the array. This information is stored in the descriptor part of a slow elements dictionary triplet. It is important to note that Array functions perform

like `Object(o.f)()`. @bmeurer patched V8 to make that pattern as fast as `const f = o.f; f()`.

Aug 31, 2017

V8
@v8js

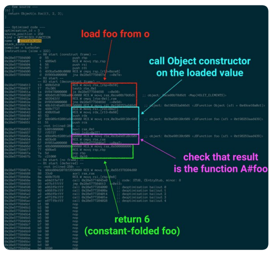
TurboFan now folds away calls like `Object(value)` when `value` is already a JavaScript object. This speeds up webpack v3-generated code.

Aug 31, 2017

V8 Retweeted

Benedikt Meurer
@bmeurer

New blog post "Object constructor calls in webpack bundles" with some background on the recent webpack findings
benediktmeurer.de/2017/08/31/obj…


load foo from o
call Object constructor on the loaded value
check that result is the function A#foo
return 6 (constant-folded foo)

Aug 31, 2017

V8
@v8js

Want to learn something about V8 internals? @camillobruni explains how V8 stores and optimizes properties:
v8project.blogspot.com/2017/08/fast-p…

Aug 30, 2017

V8
@v8js

The V8 project now has its own Twitter account. Follow @v8js for the latest updates from the world of JavaScript engines! 🚀

Aug 29, 2017

V8 Retweeted

Pony Foo
@ponyfoo

📰 Fresh content!
💝 Investigating Performance of Object#toString in ES2015
✍ By @bmeurer
🔄
ponyfoo.com/articles/inves…

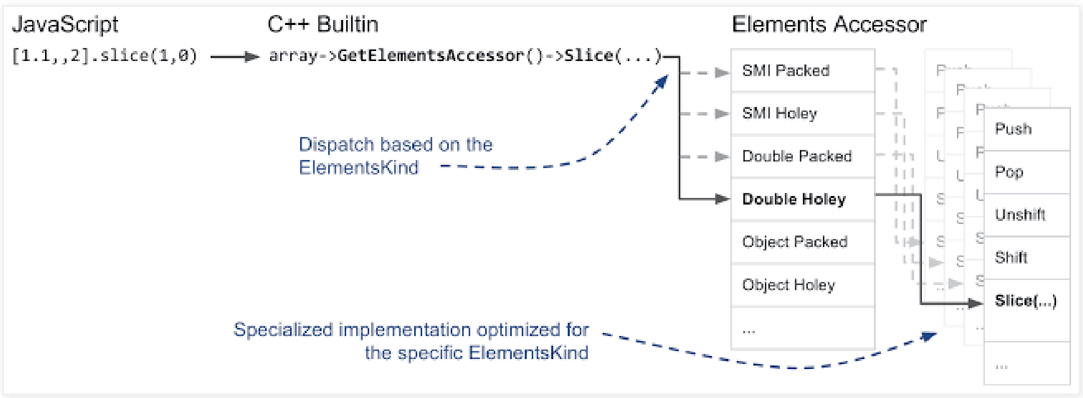
considerably slower on objects with slow elements.

Smi and Double Elements: For fast elements there is another important distinction made in V8. For instance if you only store integers in an Array, a common use-case, the GC does not have to look at the array, as integers are directly encoded as so called small integers (Smis) in place. Another special case are Arrays that only contain doubles. Unlike Smis, floating point numbers are usually represented as full objects occupying several words. However, V8 stores raw doubles for pure double arrays to avoid memory and performance overhead. The following example lists 4 examples of Smi and double elements:

```
const a1 = [1,   2, 3];  // Smi Packed
const a2 = [1,    , 3];  // Smi Holey, a2[1] reads from the prototype
const b1 = [1.1, 2, 3];  // Double Packed
const b2 = [1.1,  , 3];  // Double Holey, b2[1] reads from the prototype
```

Special Elements: With the information so far we covered 7 out of the 20 different element kinds. For simplicity we excluded 9 element kinds for TypedArrays, two more for String wrappers and last but not least, two more special element kinds for arguments objects.

The ElementsAccessor: As you can imagine we are not exactly keen on writing Array functions 20 times in C++, once for every elements kind. That's where some C++ magic comes into play. Instead of implementing Array functions over and over again, we built the ElementsAccessor where we mostly have to implement only simple functions that access elements from the backing store. The ElementsAccessor relies on CRTP to create specialized versions of each Array function. So if you call something like slice on a array, V8 internally calls a builtin written in C++ and dispatches through the ElementsAccessor to the specialized version of the function:



Takeaway from this section:

- There are fast and dictionary-mode indexed properties and elements.
- Fast properties can be packed or they can can contain holes which indicate that an indexed property has been deleted.
- Elements are specialized on their content to speed up Array functions and reduce GC overhead.

Understanding how properties work is key to many optimizations in V8. For JavaScript developers many of these internal decisions are not visible directly, but they explain why certain code patterns are faster than others. Changing the property or element type typically causes V8 to create a different HiddenClass which can lead to type pollution which prevents V8 from generating optimal code. Stay tuned for further posts on how the VM-internals of V8 work.

Posted by Camillo Bruni (@camillobruni), also author of “Fast `for-in`”

Posted by Camillo Bruni at 5:29 AM

G+

5 comments:

skowronkow August 31, 2017 at 11:43 AM

thanks!

Reply

NAITIK TYAGI September 1, 2017 at 10:59 AM

Thank you

Reply

Investigating Perfor…
We discuss how Objec…
ponyfoo.com

Aug 14, 2017

V8 Retweeted

Franziska Hinkelm…
@fhinkel

Interested in learning a bit about V8's bytecode?
medium.com/dailyjs/unders…

Understanding V8's …
V8 is Google's open s…
medium.com

Aug 17, 2017

V8 Retweeted

Franziska Hinkelm…
@fhinkel

Many of you asked for more resources on V8 and how to to contribute. So I wrote up my answer. #jschannel medium.com/@fhinkel/how-d…

How do I get started …
Are you interested in u…
medium.com

Jul 30, 2017

V8 Retweeted

Mathias Bynens
@mathias

v8project.blogspot.com/2017/08/about-… @hashseed explains the Node.js hash flooding vulnerability, and how V8 and Node.js have evolved since.

Aug 12, 2017

Load more Tweets

Embed View on Twitter

Rezoner Sikorski September 2, 2017 at 1:12 AM

Thanks for a write up. It's a good addition to mr.Raleph article about megamorphism.

Reply

Venkat September 8, 2017 at 7:25 PM

Thanks for the explantions.

Reply

Airinor September 9, 2017 at 5:37 PM

Thanks for the article! Btw, which software did you use to make this pictures?

Reply

Enter your comment...

Comment as: Select profile...

Publish Preview