Frontend Almanac
@frontend_almanac

March 31

Object structure in JavaScript engines

From a developer's perspective, objects in JavaScript are quite flexible and understandable. We can add, remove, and modify object properties on our own. However, few people think about how objects are stored in memory and processed by JS engines. Can a developer's actions, directly or indirectly, impact performance and memory consumption? Let's try to delve into all of this in this article.

Object and its properties

Before delving into the internal structures of the object, let's quickly review the basics and recall what an object actually is. The ECMA-262 specification in section 6.1.7 The Object Type defines an object rather primitively, simply as a set of properties. Object properties are represented as a "key-value" structure, where the key is the property name, and the value is a set of attributes. All object properties can be conventionally divided into two types: **data properties** and **accessor properties**.

Data properties

Properties which have the following attributes:

- `[[Value]]` - the value of the property
- `[[Writable]]` - *boolean*, by default set to false - if false, the [[Value]] cannot be changed
- `[[Enumerable]]` - *boolean*, by default set to false - if true, the property can be iterated through using "for-in"
- `[[Configurable]]` - *boolean*, by default set to false - if false, the property cannot be deleted, its type cannot be changed from Data property to Accessor

property (or vice versa), no attributes except for `[[Value]]` and setting `[[Writable]]` can be set to false

Accessor properties

Properties which have the following attributes:

- `[[Get]]` - a function that returns the value of the object
- `[[Set]]` - a function called when an attempt to assign a value to the property is made
- `[[Enumerable]]` - identical to Data property
- `[[Configurable]]` - identical to Data property.

Hidden Classes

According to the specification, in addition to the values themselves, each object property should store information about its attributes.

```
const obj1 = { a: 1, b: 2 };
```

The simple object mentioned above, in the context of a JavaScript engine, should look something like this.

```
{
  a: {
    [[Value]]: 1,
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  b: {
    [[Value]]: 2,
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}
```

Now, let's imagine that we have two objects with similar structures.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { a: 3, b: 4 };
```

Based on the above, we need to store information about each of the four properties of these two objects. It sounds somewhat wasteful in terms of memory

consumption. Furthermore, it is evident that the configuration of these properties is the same, except for the property name and its `[[Value]]`.

The popular JS engines solve this problem using so-called **hidden classes**. This concept is often encountered in various publications and documentation. However, it intersects somewhat with the concept of JavaScript classes, so engine developers have adopted their own definitions. For example, in V8, hidden classes are referred to as **Maps** (which also intersects with the concept of JavaScript Maps). In the Chakra engine used in the Internet Explorer browser, the term **Types** is applied. Safari developers, in their JavaScriptCore engine, use the notion of **Structures**. In the SpiderMonkey engine for Mozilla, hidden classes are called **Shapes**. Actually, this term is quite popular and often appears in publications because it is unique and can hardly be confused with anything else in JavaScript.

In general, there are many interesting publications about hidden classes in the network. In particular, I recommend taking a look at Mathias Bynens' post, one of the developers of V8 and Chrome DevTools.

So, the essence of hidden classes lies in extracting meta-information and object properties into separate, reusable objects and binding such a class to the real object by reference.

In this concept, the example above can be represented as follows. Later, we will see how real Maps look in the V8 engine, but for now, I will illustrate it in a hypothetical way.

```
Map1 {
  a: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  b: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

ob1 {
  map: Map1,
  values: { a: 1, a: 2 }
}

ob2 {
```

```
  map: Map1,
  values: { a: 3, a: 4 }
}
```

Hidden Classes Inheritance

The concept of hidden classes looks good in the case of objects with the same shape. However, what to do if the second object has a different structure? In the following example, the two objects are not structurally identical to each other, but have an intersection.

```
const obj1 = { a: 1, b: 2 };
const obj2 = { a: 3, b: 4, c: 5 };
```

According to the logic described above, two classes with different shapes should appear in memory. However, this leads back to the issue of attribute duplication. To avoid this, it is accepted to inherit hidden classes from each other.

```
Map1 {
  a: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  b: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

Map2 {
  back_pointer: Map1,
  c: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

ob1 {
  map: Map1,
  values: { a: 1, b: 2 }
}

ob2 {
```

```
  map: Map2,
  values: { a: 3, b: 4, c: 5 }
}
```

Here we see that the class `Map2` describes only one property and a reference to an object with a more "specific" shape.

It is also worth mentioning that the shape of the hidden class is influenced not only by the set of properties but also by their order. In other words, the following objects will have different shapes of hidden classes.

```
Map1 {
  a: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  b: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

Map2 {
  b: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  a: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

ob1 {
  map: Map1,
  values: { a: 1, b: 2 }
}

ob2 {
  map: Map2,
  values: { b: 3, a: 4 }
}
```

If we change the shape of the object after initialization, this also leads to the creation of a new hidden subclass.

```
const ob1 = { a: 1, b: 2 };
obj1.c = 3;

const obj2 = { a: 4, b: 5, c: 6 };
```

This example results in the following structure of hidden classes.

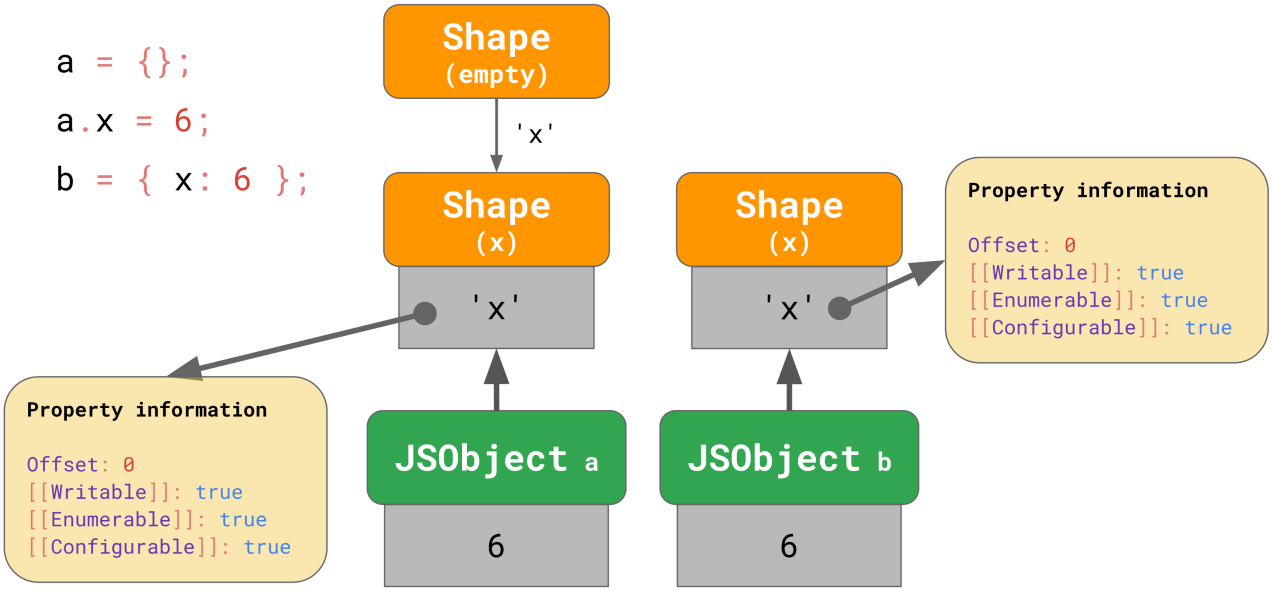
```
Map1 {
  a: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  },
  b: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

Map2 {
  back_pointer: Map1,
  c: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

Map3 {
  back_pointer: Map1,
  c: {
    [[Writable]]: true,
    [[Enumerable]]: true,
    [[Configurable]]: true,
  }
}

ob1 {
  map: Map2,
  values: { a: 1, b: 2, c: 3 }
}
```

```
ob2 {
  map: Map3,
  values: { a: 4, b: 5, c: 6 }
}
```

Hidden Classes in Practice

A bit earlier, I referred to Mathias Binnen's post on object shapes. However, many years have passed since then. For the sake of the experiment's purity, I decided to check the practical situation in the real V8 engine.

Let's conduct an experiment using the example provided in Mathias's article.



To do this, we will need the V8's built-in internal method - `%DebugPrint`. Just a reminder, in order to use the engine's built-in methods, it needs to be launched with the `--allow-natives-syntax` flag. To see detailed information about JS objects, the engine must be compiled in `debug` mode.

```
d8> const a = {};
d8> a.x = 6;
d8> const b = { x: 6 };
d8>
d8> %DebugPrint(a);
DebugPrint: 0x1d47001c9425: [JS_OBJECT_TYPE]
 - map: 0x1d47000da9a9 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
 - elements: 0x1d47000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x1d47000006cd <FixedArray[0]>
 - All own properties (excluding elements): {
    0x1d4700002b91: [String] in ReadOnlySpace: #x: 6 (const data field 0),
```

```
 }
0x1d47000da9a9: [Map] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 3
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x1d47000c4945 <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x1d47000da9f1 <Cell value= 0>
 - instance descriptors (own) #1: 0x1d47001cb111 <DescriptorArray[1]>
 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
 - constructor: 0x1d47000c4655 <JSFunction Object (sfi = 0x1d4700335385)>
 - dependent code: 0x1d47000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0
```

We see an object `a` located at the address `0x1d47001c9425`. The object is associated with a hidden class at the address `0x1d47000da9a9`. Inside the object itself, the value `#x: 6` is stored. The property attributes are located in the associated hidden class under the field `instance descriptors`. Just in case, let's take a look at the array of descriptors at the specified address.

```
d8> %DebugPrintPtr(0x1d47001cb111)
DebugPrint: 0x1d47001cb111: [DescriptorArray]
 - map: 0x1d470000062d <Map(DESCRIPTOR_ARRAY_TYPE)>
 - enum_cache: 1
   - keys: 0x1d47000dacad <FixedArray[1]>
   - indices: 0x1d47000dacb9 <FixedArray[1]>
 - nof slack descriptors: 0
 - nof descriptors: 1
 - raw gc state: mc epoch 0, marked 0, delta 0
  [0]: 0x1d4700002b91: [String] in ReadOnlySpace: #x (const data field 0:s
0x1d470000062d: [Map] in ReadOnlySpace
 - map: 0x1d47000004c5 <MetaMap (0x1d470000007d <null>)>
 - type: DESCRIPTOR_ARRAY_TYPE
 - instance size: variable
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - non-extensible
 - back pointer: 0x1d4700000061 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x1d4700000701 <DescriptorArray[0]>
 - prototype: 0x1d470000007d <null>
```

```
 - constructor: 0x1d470000007d <null>
 - dependent code: 0x1d47000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

32190781763857
```

The array of descriptors contains an element `#x`, which holds all the necessary information about the object property.

Now let's take a look at the `back pointer` link with the address `0x1d47000c4945`.

```
d8> %DebugPrintPtr(0x1d47000c4945)
DebugPrint: 0x1d47000c4945: [Map] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 4
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - back pointer: 0x1d4700000061 <undefined>
 - prototype_validity cell: 0x1d4700000a31 <Cell value= 1>
 - instance descriptors (own) #0: 0x1d4700000701 <DescriptorArray[0]>
 - transitions #1: 0x1d47000da9d1 <TransitionArray[6]>Transition array #1:
     0x1d4700002b91: [String] in ReadOnlySpace: #x: (transition to (const

 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
 - constructor: 0x1d47000c4655 <JSFunction Object (sfi = 0x1d4700335385)>
 - dependent code: 0x1d47000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0
0x1d47000c3c29: [MetaMap] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: MAP_TYPE
 - instance size: 40
 - native_context: 0x1d47000c3c79 <NativeContext[285]>

32190780688709
```

This hidden class represents an empty object. Its array of descriptors is empty, and the `back pointer` reference is not defined.

Now let's take a look at object `b`.

```
d8> %DebugPrint(b)
DebugPrint: 0x1d47001cb169: [JS_OBJECT_TYPE]
 - map: 0x1d47000dab39 <Map[16](HOLEY_ELEMENTS)> [FastProperties]
```

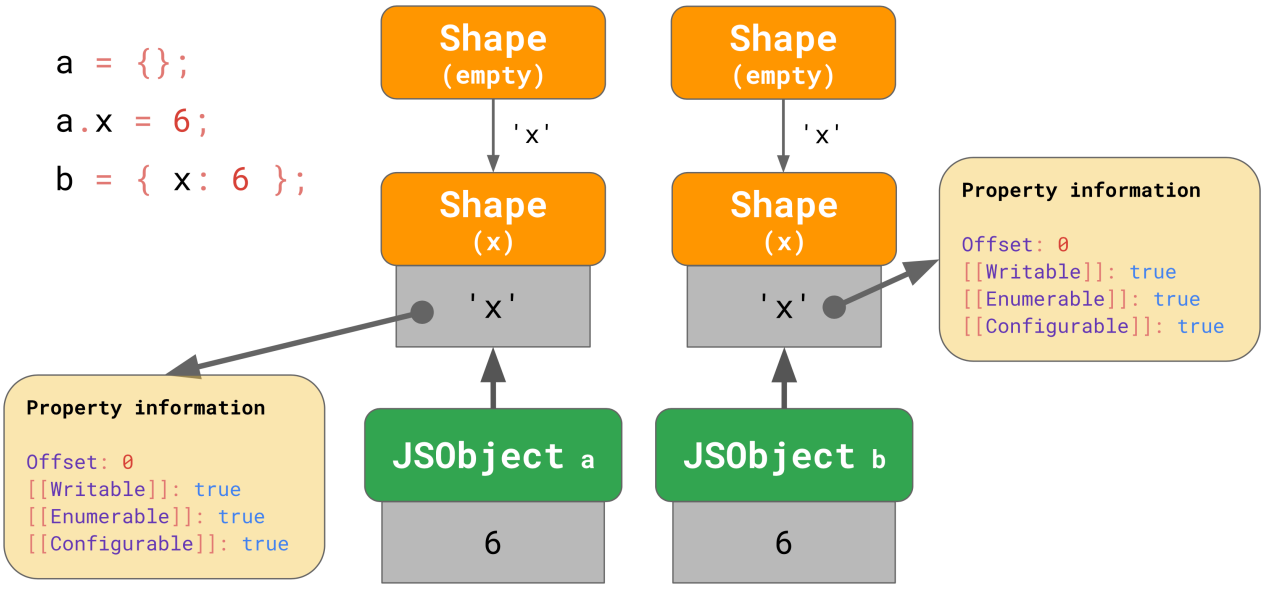
```
 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
 - elements: 0x1d47000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x1d47000006cd <FixedArray[0]>
 - All own properties (excluding elements): {
    0x1d4700002b91: [String] in ReadOnlySpace: #x: 6 (const data field 0),
 }
0x1d47000dab39: [Map] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - unused property fields: 0
 - elements kind: HOLEY_ELEMENTS
 - enum length: 1
 - stable_map
 - back pointer: 0x1d47000dab11 <Map[16](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x1d4700000a31 <Cell value= 1>
 - instance descriptors (own) #1: 0x1d47001cb179 <DescriptorArray[1]>
 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
 - constructor: 0x1d47000c4655 <JSFunction Object (sfi = 0x1d4700335385)>
 - dependent code: 0x1d47000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

{x: 6}
```

The value of the property is also stored within the object itself, while the attributes of the property are stored in an array of descriptors of the hidden class. However, I would like to point out that the `back pointer` reference here is not empty either, although it should not be present in the parent class diagram provided. Let's take a look at the class through this reference.

```
d8> %DebugPrintPtr(0x1d47000dab11)
DebugPrint: 0x1d47000dab11: [Map] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - unused property fields: 1
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - back pointer: 0x1d4700000061 <undefined>
 - prototype_validity cell: 0x1d4700000a31 <Cell value= 1>
 - instance descriptors (own) #0: 0x1d4700000701 <DescriptorArray[0]>
 - transitions #1: 0x1d47000dab39 <Map[16](HOLEY_ELEMENTS)>
     0x1d4700002b91: [String] in ReadOnlySpace: #x: (transition to (const
 - prototype: 0x1d47000c4b11 <Object map = 0x1d47000c414d>
```

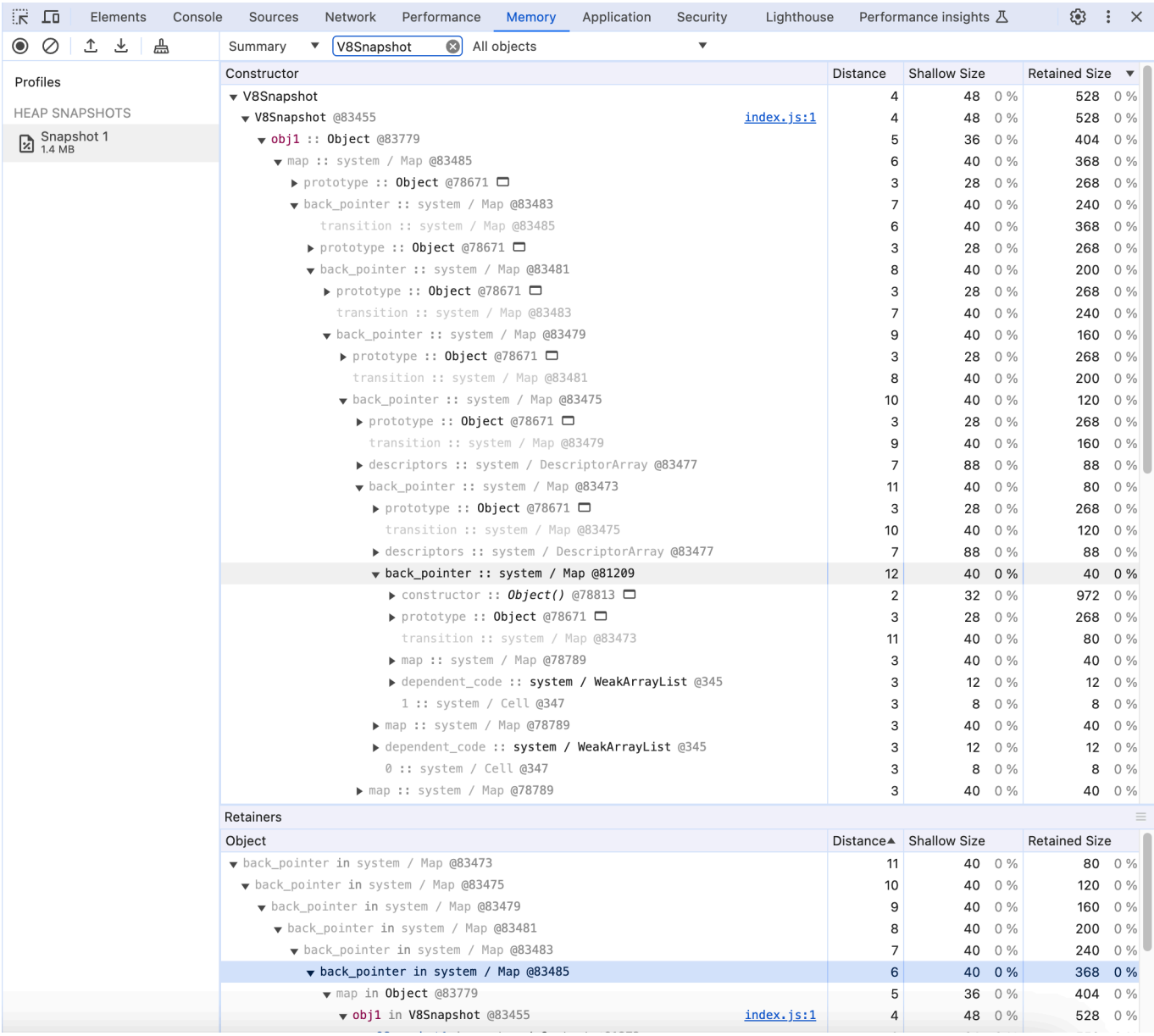
```
 - constructor: 0x1d47000c4655 <JSFunction Object (sfi = 0x1d4700335385)>
 - dependent code: 0x1d47000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0
0x1d47000c3c29: [MetaMap] in OldSpace
 - map: 0x1d47000c3c29 <MetaMap (0x1d47000c3c79 <NativeContext[285]>)>
 - type: MAP_TYPE
 - instance size: 40
 - native_context: 0x1d47000c3c79 <NativeContext[285]>

32190780779281
```

The class looks exactly like the hidden class of an empty object above, but with a different address. This means that it is, in fact, a duplicate of the previous class. Therefore, the actual structure of this example looks as follows.



This is the first deviation from the theory. To understand the need for another hidden class for an empty object, we will need an object with multiple properties. Let's assume that the original object initially has several properties. It will not be very convenient to explore such an object through a command line, so let's use Chrome DevTools. For convenience, we will enclose the object within the function context.

```
function V8Snapshot() {
  this.obj1 = { a: 1, b: 2, c: 3, d: 4, e: 5, f: 6 };
}

const v8Snapshot1 = new V8Snapshot();
```



The memory snapshot shows 6 inherited classes for this object, which equals the number of object properties. This is the second deviation from the theory, according to which it was assumed that the object initially has a single hidden class, the shape of which contains the properties with which it was initialized. The reason for this lies in the fact that in practice we operate not with a single object, but with several, perhaps even tens, hundreds, or thousands. In such circumstances, searching for and restructuring class trees can be quite expensive. So we have come to another concept of JS engines.

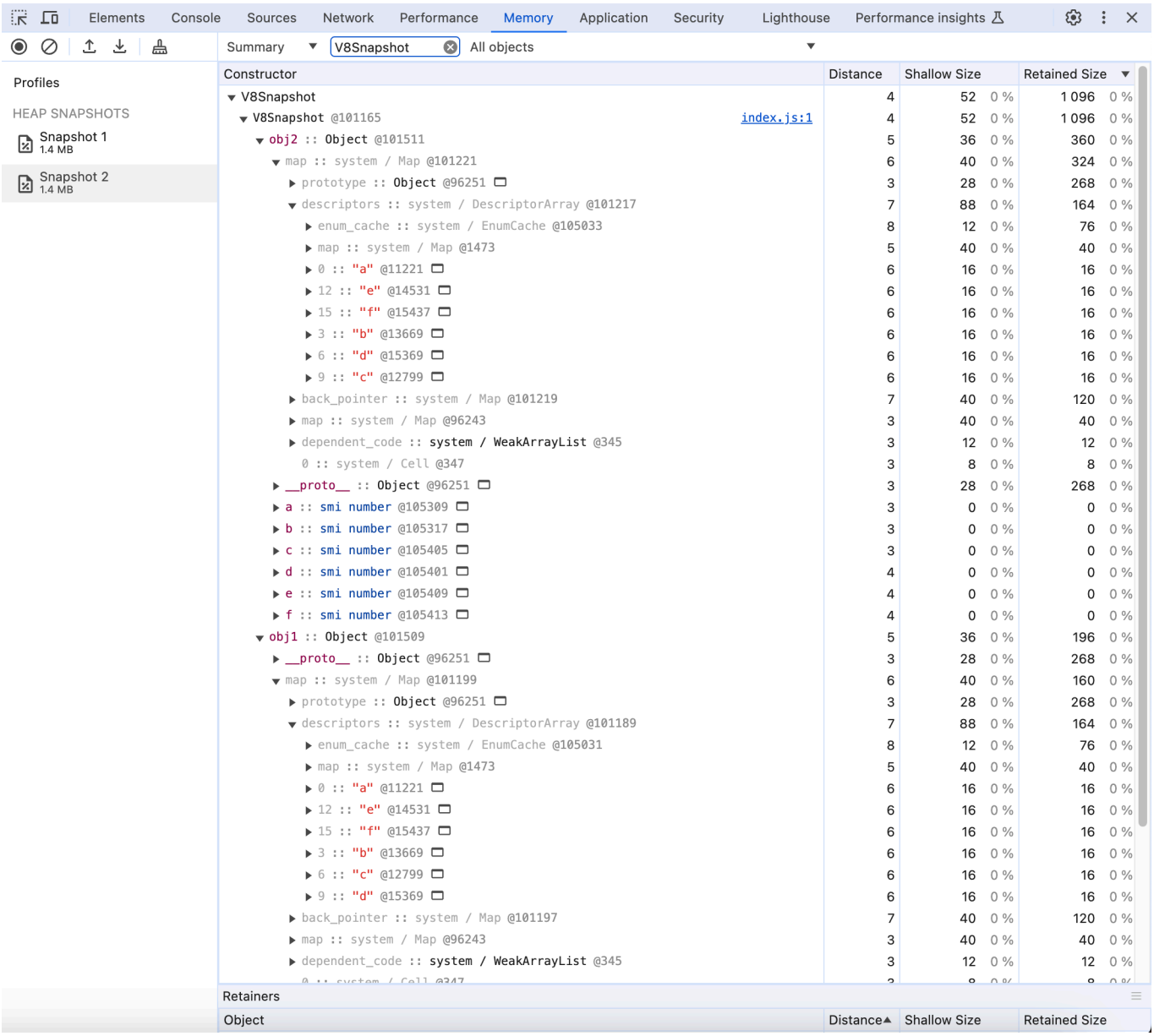
Transitions

Let's add another object with a similar shape to the example above.

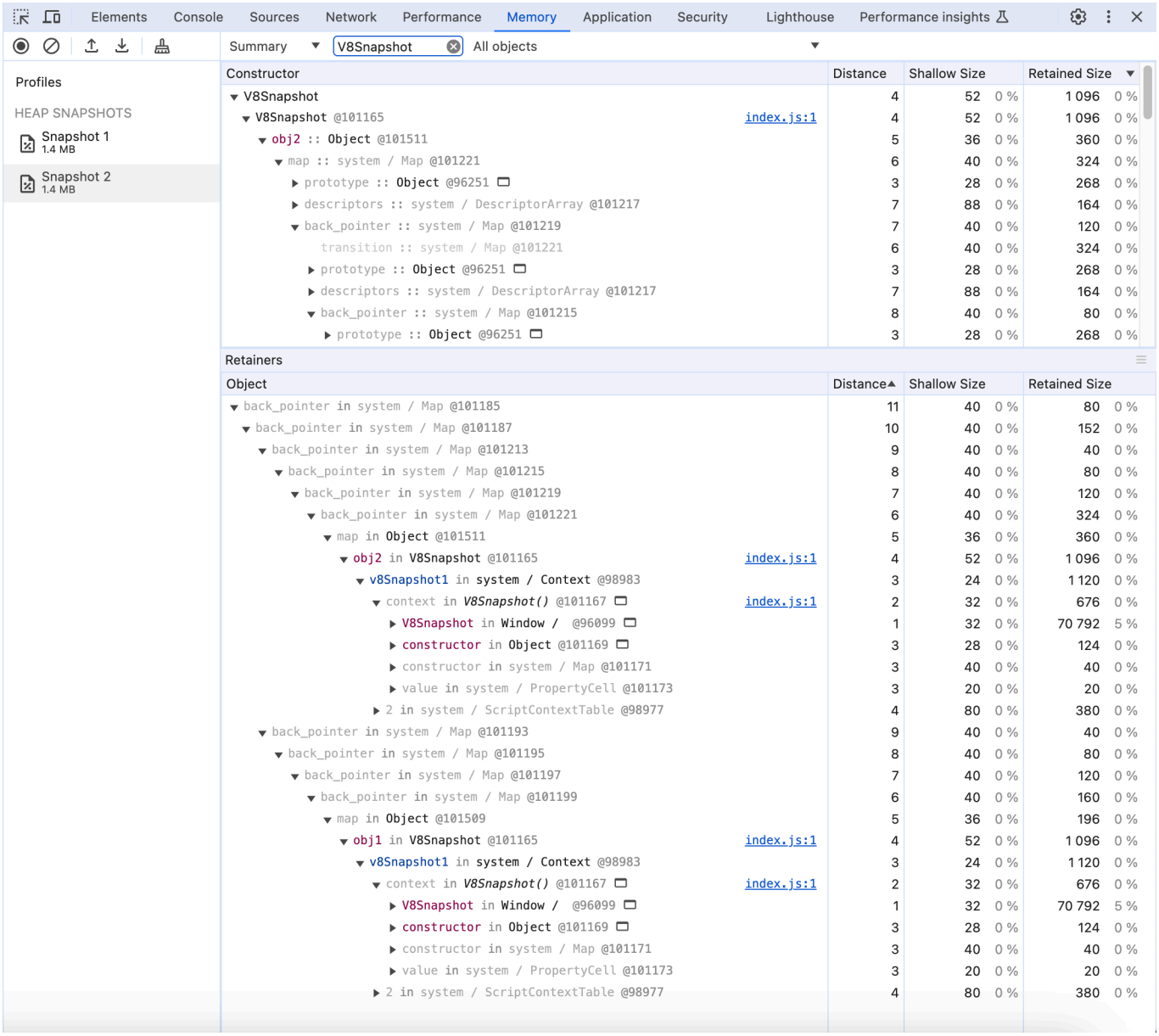
```
function V8Snapshot() {
  this.obj1 = { a: 1, b: 2, c: 3, d: 4, e: 5, f: 6 };
  this.obj2 = { a: 1, b: 2, d: 3, c: 4, e: 5, f: 6 };
}
```

```
const v8Snapshot1 = new V8Snapshot();
```

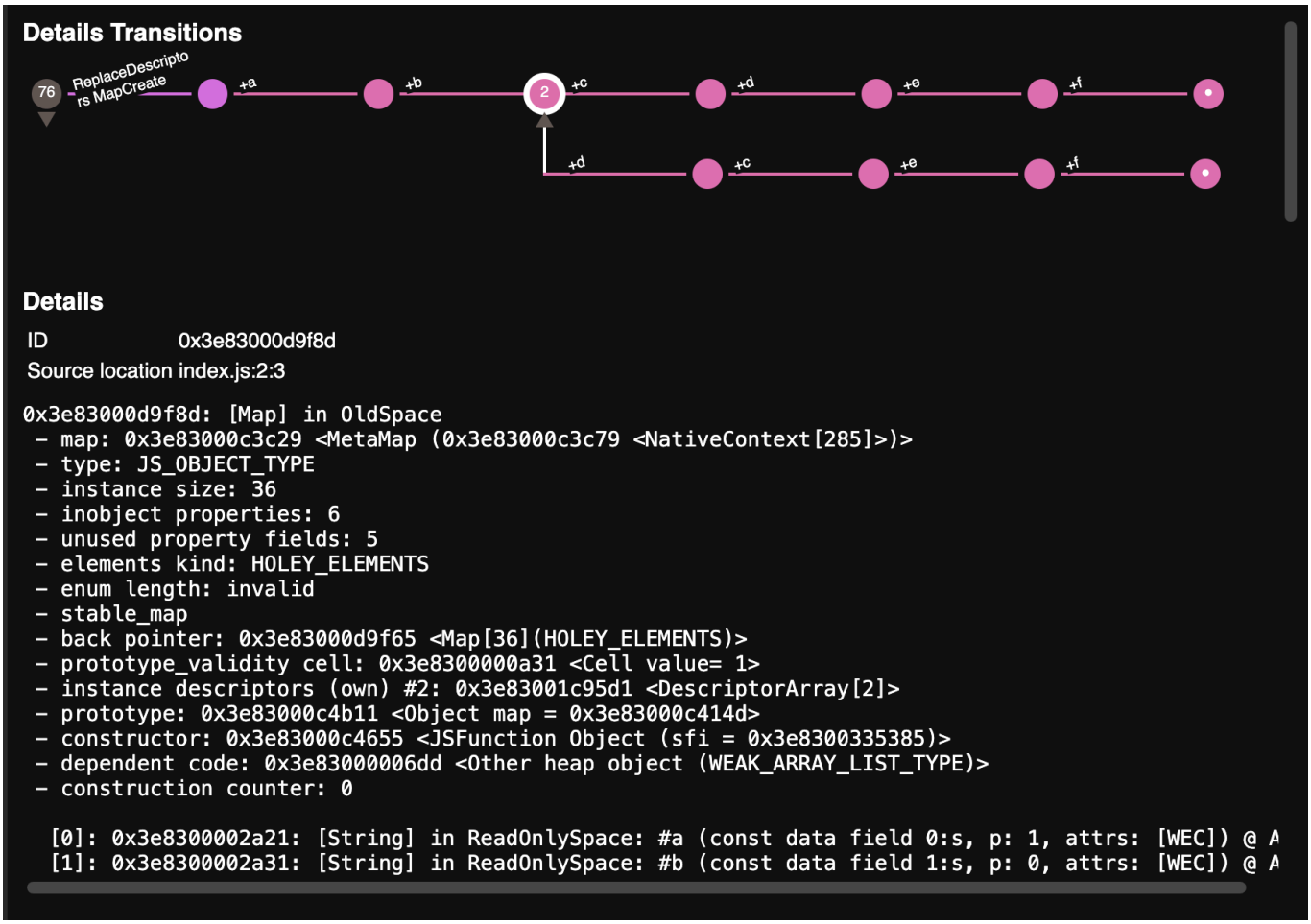
At first glance, the shape of the second object is very similar, but the properties `c` and `d` have a different order.



In the descriptor arrays, these properties will have different indexes. The class with the address `@101187` has two descendants.



For better clarity, let's run the script log through the V8 System Analyzer.



It is clearly visible here that the original form `{ a, b, c, d, e, f }` extends at the point `c`. However, the interpreter does not recognize this until it starts initializing the second object. In order to create a new class tree, the engine would have to search for a class in the heap that matches the form, break it down into parts, create new classes, and reassign them to all created objects. To avoid this, the developers of V8 decided to divide the class into a set of minimal forms right away, starting with an empty class during the first object initialization.

```
{}
{ a }
{ a, b }
{ a, b, c }
{ a, b, c, d }
{ a, b, c, d, e }
{ a, b, c, d, e, f }
```

The process of creating a new hidden class with the addition or modification of any property is called a **transition**. In our case, the first object will have 6 transitions initially (+a, +b, +c, etc.).

This approach allows for the following: a) easily find a suitable initial form for the new object, b) there is no need to rebuild anything, just create a new class with a reference to the appropriate minimal form.

```
                {}
                { a }
                { a, b }

{ a, b, c }             { a, b, d }
{ a, b, c, d }          { a, b, d, c }
{ a, b, c, d, e }       { a, b, d, c, e }
{ a, b, c, d, e, f }    { a, b, d, c, e, f }
```

In-object and External Properties.

Let's consider the following example:

```
d8> const obj1 = { a: 1 };
d8> obj1.b = 2;
d8>
d8> %DebugPrint(obj1);
DebugPrint: 0×2387001c942d: [JS_OBJECT_TYPE]
 - map: 0×2387000dabb1 <Map[16](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0×2387000c4b11 <Object map = 0×2387000c414d>
 - elements: 0×2387000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0×2387001cb521 <PropertyArray[3]>
 - All own properties (excluding elements): {
    0×238700002a21: [String] in ReadOnlySpace: #a: 1 (const data field 0),
    0×238700002a31: [String] in ReadOnlySpace: #b: 2 (const data field 1),
 }
0×2387000dabb1: [Map] in OldSpace
 - map: 0×2387000c3c29 <MetaMap (0×2387000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - unused property fields: 2
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0×2387000d9ca1 <Map[16](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0×2387000dabd9 <Cell value= 0>
 - instance descriptors (own) #2: 0×2387001cb4f9 <DescriptorArray[2]>
 - prototype: 0×2387000c4b11 <Object map = 0×2387000c414d>
 - constructor: 0×2387000c4655 <JSFunction Object (sfi = 0×238700335385)>
 - dependent code: 0×2387000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

 {a: 1, b: 2}
```

Upon closer inspection of the set of values of this object, we can see that the property `a` is marked as `in-object`, while the property `b` is marked as an element of the `properties` array.

```
 - All own properties (excluding elements): {
     ... #a: 1 (const data field 0), location: in-object
     ... #b: 2 (const data field 1), location: properties[0]
  }
```

This example demonstrates that some properties are stored directly inside the object itself ("in-object"), while others are stored in an external storage of properties. This is related to the fact that according to the ECMA-262 specification, JavaScript objects do not have a fixed size. By adding or removing properties from an object, its size changes. This raises the question: how much memory should be allocated for the object? Furthermore, how can we expand the already allocated memory for the object? Developers of V8 addressed these issues as follows.

In-object Properties

At the moment of the object's primary initialization, the object literal has already been parsed, and the AST tree contains information about the properties indicated at the initialization moment. This set of properties is placed directly inside the object, allowing them to be accessed quickly and with minimal overhead. These properties are referred to as **in-object**.

Let's take another look at the class of an empty object.

```
d8> const obj1 = {}
d8>
d8> %DebugPrint(obj1);
DebugPrint: 0×2d56001c9ed1: [JS_OBJECT_TYPE]
 - map: 0×2d56000c4945 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0×2d56000c4b11 <Object map = 0×2d56000c414d>
 - elements: 0×2d56000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0×2d56000006cd <FixedArray[0]>
 - All own properties (excluding elements): {}
0×2d56000c4945: [Map] in OldSpace
 - map: 0×2d56000c3c29 <MetaMap (0×2d56000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 4
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - back pointer: 0×2d5600000061 <undefined>
```

```
 - prototype_validity cell: 0x2d5600000a31 <Cell value= 1>
 - instance descriptors (own) #0: 0x2d5600000701 <DescriptorArray[0]>
 - prototype: 0x2d56000c4b11 <Object map = 0x2d56000c414d>
 - constructor: 0x2d56000c4655 <JSFunction Object (sfi = 0x2d5600335385)>
 - dependent code: 0x2d56000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0
```

I would like to draw your attention to the parameter `inobject properties`. Here it is set to 4, even though the object does not have any properties yet. The thing is, empty objects by default have several slots for `in-object` properties. In V8, the number of such slots is 4.

```
d8> obj1.a = 1;
d8> obj1.b = 2;
d8> obj1.c = 3;
d8> obj1.d = 4;
d8> obj1.e = 5;
d8> obj1.f = 6;
d8>
d8> %DebugPrint(obj1);
DebugPrint: 0x2d56001c9ed1: [JS_OBJECT_TYPE]
 - map: 0x2d56000db291 <Map[28](HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x2d56000c4b11 <Object map = 0x2d56000c414d>
 - elements: 0x2d56000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x2d56001cc1a9 <PropertyArray[3]>
 - All own properties (excluding elements): {
    0x2d5600002a21: [String] in ReadOnlySpace: #a: 1 (const data field 0),
    0x2d5600002a31: [String] in ReadOnlySpace: #b: 2 (const data field 1),
    0x2d5600002a41: [String] in ReadOnlySpace: #c: 3 (const data field 2),
    0x2d5600002a51: [String] in ReadOnlySpace: #d: 4 (const data field 3),
    0x2d5600002a61: [String] in ReadOnlySpace: #e: 5 (const data field 4),
    0x2d5600002a71: [String] in ReadOnlySpace: #f: 6 (const data field 5),
 }
0x2d56000db291: [Map] in OldSpace
 - map: 0x2d56000c3c29 <MetaMap (0x2d56000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 28
 - inobject properties: 4
 - unused property fields: 1
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x2d56000db169 <Map[28](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x2d56000dace9 <Cell value= 0>
 - instance descriptors (own) #6: 0x2d56001cc1f5 <DescriptorArray[6]>
 - prototype: 0x2d56000c4b11 <Object map = 0x2d56000c414d>
```

```
 - constructor: 0x2d56000c4655 <JSFunction Object (sfi = 0x2d5600335385)>
 - dependent code: 0x2d56000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0
```

This means that the first 4 properties added to an empty object will be placed in these slots as `in-object` properties.

External Properties

Properties that were added after initialization can no longer be placed inside the object since memory for the object is already allocated. To avoid wasting resources on reallocating the entire object, the engine places such properties in an external storage, in this case, in an external array of properties, a reference to which already exists inside the object. These properties are called **external** or **normal** (this exact term can often be found in publications by V8 developers). Access to such properties is slightly slower, as it requires resolving the reference to the storage and obtaining the property by index. However, this is much more efficient than reallocating the entire object.

Fast and Slow Properties

The external property from the example above, as we have just discussed, is stored in an external property array directly linked to our object. The data format in this array is identical to the format of internal properties. In other words, only property values are stored there, while metadata about them is placed in the descriptors array, which also contains information about internal properties. Essentially, external properties differ from internal ones only in the location where they are stored. Both can be considered fast properties in a broad sense. However, I would like to remind you that JavaScript is a dynamic and flexible programming language. A developer has the ability to add, remove, and modify object properties as desired. Active changes to the set of properties can lead to significant processor time costs. To optimize this process, V8 supports the so-called "slow" properties. The essence of slow properties lies in using a different type of external storage. Instead of an array of values, properties are placed in a separate dictionary object together with all their attributes. Access to both the values and attributes of such properties is done by their name, which serves as the key to the dictionary.

```
d8> delete obj1.a;
d8>
d8> %DebugPrint(obj1)
DebugPrint: 0x2387001c942d: [JS_OBJECT_TYPE]
 - map: 0x2387000d6071 <Map[12](HOLEY_ELEMENTS)> [DictionaryProperties]
 - prototype: 0x2387000c4b11 <Object map = 0x2387000c414d>
 - elements: 0x2387000006cd <FixedArray[0]> [HOLEY_ELEMENTS]
```

```
 - properties: 0x2387001cc1d9 <NameDictionary[30]>
 - All own properties (excluding elements): {
   b: 2 (data, dict_index: 2, attrs: [WEC])
 }
0x2387000d6071: [Map] in OldSpace
 - map: 0x2387000c3c29 <MetaMap (0x2387000c3c79 <NativeContext[285]>)>
 - type: JS_OBJECT_TYPE
 - instance size: 12
 - inobject properties: 0
 - unused property fields: 0
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - dictionary_map
 - may_have_interesting_properties
 - back pointer: 0x238700000061 <undefined>
 - prototype_validity cell: 0x238700000a31 <Cell value= 1>
 - instance descriptors (own) #0: 0x238700000701 <DescriptorArray[0]>
 - prototype: 0x2387000c4b11 <Object map = 0x2387000c414d>
 - constructor: 0x2387000c4655 <JSFunction Object (sfi = 0x238700335385)>
 - dependent code: 0x2387000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

{b: 2}
```

We have deleted the property `obj1.a`. Despite the fact that the property was internal, we completely changed the shape of the hidden class. To be precise, we have shrunk it, which is different from the typical shape extension. This means that the tree of shpaes has become shorter; hence, the descriptors and value arrays must also be reconstructed. All these operations require certain time resources. In order to avoid this, the engine changes the way object properties are stored to a slower method using an object dictionary. In this example, the dictionary (`NameDictionary`) is located at address `0x2387001cc1d9`.

```
d8> %DebugPrintPtr(0x2387001cc1d9)
DebugPrint: 0x2387001cc1d9: [NameDictionary]
 - FixedArray length: 30
 - elements: 1
 - deleted: 1
 - capacity: 8
 - elements: {
              7: b → 2 (data, dict_index: 2, attrs: [WEC])
 }
0x238700000ba1: [Map] in ReadOnlySpace
 - map: 0x2387000004c5 <MetaMap (0x23870000007d <null>)>
 - type: NAME_DICTIONARY_TYPE
 - instance size: variable
```

```
 - elements kind: HOLEY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x238700000061 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x238700000701 <DescriptorArray[0]>
 - prototype: 0x23870000007d <null>
 - constructor: 0x23870000007d <null>
 - dependent code: 0x2387000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

39062729441753
```

Arrays

According to the 23.1 Array Objects section of the specification, an array is an object whose keys are integers from `0` to `2**32 - 2`. On the one hand, it seems that from the perspective of hidden classes, an array is no different from a regular object. However, in practice, arrays can be quite large. What if there are thousands of elements in an array? Will a separate hidden class be created for each element? Let's see what the hidden class of an array actually looks like.

```
d8> arr = [];
d8> arr[0] = 1;
d8> arr[1] = 2;
d8>
d8> %DebugPrint(arr);
DebugPrint: 0x24001c9421: [JSArray]
 - map: 0x0024000ce6b1 <Map[16](PACKED_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x0024000ce925 <JSArray[0]>
 - elements: 0x0024001cb125 <FixedArray[17]> [PACKED_SMI_ELEMENTS]
 - length: 2
 - properties: 0x0024000006cd <FixedArray[0]>
 - All own properties (excluding elements): {
    0x2400000d41: [String] in ReadOnlySpace: #length: 0x00240030f6f9 <Acce
 }
 - elements: 0x0024001cb125 <FixedArray[17]> {
           0: 1
           1: 2
       2-16: 0x0024000006e9 <the_hole_value>
 }
0x24000ce6b1: [Map] in OldSpace
 - map: 0x0024000c3c29 <MetaMap (0x0024000c3c79 <NativeContext[285]>)>
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
```

```
 - unused property fields: 0
 - elements kind: PACKED_SMI_ELEMENTS
 - enum length: invalid
 - back pointer: 0x002400000061 <undefined>
 - prototype_validity cell: 0x002400000a31 <Cell value= 1>
 - instance descriptors #1: 0x0024000cef3d <DescriptorArray[1]>
 - transitions #1: 0x0024000cef59 <TransitionArray[4]>Transition array #1:
     0x002400000e05 <Symbol: (elements_transition_symbol)>: (transition to

 - prototype: 0x0024000ce925 <JSArray[0]>
 - constructor: 0x0024000ce61d <JSFunction Array (sfi = 0x2400335da5)>
 - dependent code: 0x0024000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

[1, 2]
```

As we can see, in the hidden class of this object, the `back pointer` reference is empty, indicating the absence of a parent class, even though we have added two elements. The thing is, the hidden class of any array always has a uniform shape of `JS_ARRAY_TYPE`. This is a special hidden class that only has one property in its descriptors - `length`. The array elements, on the other hand, are arranged inside the object in a `FixedArray` structure. In reality, hidden array classes can still be inherited, as the elements themselves can have different data types, and keys, depending on the number, can be stored in different ways for optimization of access to them. In this article, I will not delve into all the possible transitions within arrays in detail, as this is a topic for a separate article. However, it is worth noting that various non-standard manipulations with array keys can lead to the creation of a class tree for all or some of the elements.

```
d8> const arr = [];
d8> arr[-1] = 1;
d8> arr[2**32 - 1] = 2;
d8>
d8> %DebugPrint(arr)
DebugPrint: 0x0e0b001c98c9: [JSArray]
 - map: 0x0e0b000dacc1 <Map[16](PACKED_SMI_ELEMENTS)> [FastProperties]
 - prototype: 0x0e0b000ce925 <JSArray[0]>
 - elements: 0x0e0b000006cd <FixedArray[0]> [PACKED_SMI_ELEMENTS]
 - length: 0
 - properties: 0x0e0b001cb5f1 <PropertyArray[3]>
 - All own properties (excluding elements): {
    0x0e0b00000d41: [String] in ReadOnlySpace: #length: 0x0e0b0030f6f9 <Acc
    0x0e0b000dab35: [String] in OldSpace: #-1: 1 (const data field 0), loca
    0x0e0b000daca9: [String] in OldSpace: #4294967295: 2 (const data field
 }
```

```
0x2e0b000dacc1: [Map] in OldSpace
 - map: 0x0e0b000c3c29 <MetaMap (0x0e0b000c3c79 <NativeContext[285]>)>
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - unused property fields: 1
 - elements kind: PACKED_SMI_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x0e0b000dab45 <Map[16](PACKED_SMI_ELEMENTS)>
 - prototype_validity cell: 0x0e0b000dab95 <Cell value= 0>
 - instance descriptors (own) #3: 0x0e0b001cb651 <DescriptorArray[3]>
 - prototype: 0x0e0b000ce925 <JSArray[0]>
 - constructor: 0x0e0b000ce61d <JSFunction Array (sfi = 0x0e0b00335da5)>
 - dependent code: 0x0e0b000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

[]
```

In the example above, both elements `-1` and `2**32 - 1` are not within the range of possible array indexes `[0 .. 2**32 - 2]` and were declared as regular object properties with corresponding shapes and hidden class tree generation.

Another exceptional situation may occur when attempting to change the index attributes. For elements to be stored in a fast store, all indexes must have the same configuration. Trying to change the attributes of any of the indexes will not result in the creation of a separate property, but will lead to a change in the storage type to slow, in which not only values but also attributes of each index will be stored. Essentially, the same rule is applied here as with slow object properties.

```
d8> const arr = [1];
d8> Object.defineProperty(arr, '0', { value: 2, writable:  false });
d8> arr.push(3);
d8>
d8> %DebugPrint(arr);
DebugPrint: 0x29ee001c9425: [JSArray]
 - map: 0x29ee000dad05 <Map[16](DICTIONARY_ELEMENTS)> [FastProperties]
 - prototype: 0x29ee000ce925 <JSArray[0]>
 - elements: 0x29ee001cb391 <NumberDictionary[16]> [DICTIONARY_ELEMENTS]
 - length: 2
 - properties: 0x29ee000006cd <FixedArray[0]>
 - All own properties (excluding elements): {
    0x29ee00000d41: [String] in ReadOnlySpace: #length: 0x29ee0030f6f9 <Ac
 }
 - elements: 0x29ee001cb391 <NumberDictionary[16]> {
   - requires_slow_elements
```

```
    0: 2 (data, dict_index: 0, attrs: [_EC])
    1: 3 (data, dict_index: 0, attrs: [WEC])
  }
0x29ee000dad05: [Map] in OldSpace
 - map: 0x29ee000c3c29 <MetaMap (0x29ee000c3c79 <NativeContext[285]>)>
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - unused property fields: 0
 - elements kind: DICTIONARY_ELEMENTS
 - enum length: invalid
 - stable_map
 - back pointer: 0x29ee000cf071 <Map[16](HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x29ee00000a31 <Cell value= 1>
 - instance descriptors (own) #1: 0x29ee000cef3d <DescriptorArray[1]>
 - prototype: 0x29ee000ce925 <JSArray[0]>
 - constructor: 0x29ee000ce61d <JSFunction Array (sfi = 0x29ee00335da5)>
 - dependent code: 0x29ee000006dd <Other heap object (WEAK_ARRAY_LIST_TYPE
 - construction counter: 0

[2, 3]
```

Conclusion

In this article, we have delved deeper into the methods of storing object properties, concepts of hidden classes, object shapes, object descriptors, internal and external properties, as well as fast and slow methods of storing them. Let us now briefly recap the main terms and conclusions.

- Every object in JavaScript has its main internal class and a hidden class that describes its shape.
- Hidden classes inherit from each other and are organized into class trees. The shape of an object `{ a: 1 }` will be the parent for the shape of an object `{ a: 1, b: 2 }`.
- The order of properties matters. Objects `{ a: 1, b: 2 }` and `{ b: 2, a: 1 }` will have two different shapes.
- A subclass holds a reference to the superclass and information about what has changed (**transition**).
- In the class tree of each object, the number of levels is not less than the number of properties in the object.
- The fastest properties of an object will be those declared at initialization. In the following example, access to the property `obj1.a` will be faster than to

obj2.a .

```
const obj1 = { a: undefined };
obj1.a = 1; // ← "a" - in-object property

const obj2 = {};
obj2.a = 1; // ← "a" - external property
```

- Atypical changes in the object's structure, such as property removal, can lead to a change in the storage type of properties to a slower one. In the following example, `obj1` will change its type to `NamedDictionary`, and accessing its properties will be significantly slower than accessing the properties of `obj2`.

```
const obj1 = { a: 1, b: 2 };
delete obj1.a; // cheanges a storage type to NameDictionary

const obj2 = { a: 1, b: 2 };
obj2.a = undefined; // a storage type is not changed
```

- If an object has external properties but the internal ones are less 4, such an object can be slightly optimized, as an empty object by default has several slots for `in-object` properties.

```
const obj1 = { a: 1 };
obj1.b = 2;
obj1.c = 3;
obj1.d = 4;
obj1.e = 5;
obj1.f = 6;

%DebugPrint(obj1);
...
- All own properties (excluding elements): {
    ...#a: 1 (const data field 0), location: in-object
    ...#b: 2 (const data field 1), location: properties[0]
    ...#c: 3 (const data field 2), location: properties[1]
    ...#d: 4 (const data field 3), location: properties[2]
    ...#e: 5 (const data field 4), location: properties[3]
    ...#f: 6 (const data field 5), location: properties[4]
 }

const obj2 = Object.fromEntries(Object.entries(obj1));

%DebugPrint(obj2);
...
 - All own properties (excluding elements): {
```

```
    ... #a: 1 (const data field 0), location: in-object
    ... #b: 2 (const data field 1), location: in-object
    ... #c: 3 (const data field 2), location: in-object
    ... #d: 4 (const data field 3), location: in-object
    ... #e: 5 (const data field 4), location: properties[0]
    ... #f: 6 (const data field 5), location: properties[1]
}
```

- An array is a regular class whose structure looks like `{ length: [W__] }`. The elements of the array are stored in special structures, and references to these structures are placed inside the object. Adding or removing elements from the array does not lead to an increase in the class tree.

```
const arr = [];
arr[0] = 1; // new element of the array doesn't extends the shapes tree

const obj = {};
obj1[0] = 1; // each new property extends the shapes tree
```

- The use of atypical keys in an array, such as non-numeric keys or keys outside the range `[0 .. 2**32 - 2]`, leads to the creation of new shapes in the class tree.

```
const arr = [];
arr[-1] = 1;
arr[2**32 - 1] = 2;
// Leads to the shapes tree generation
// { length } ⇒ { length, [-1] } ⇒ { length, [-1], [2**32 - 1] }
```

- Attempting to modify an array element's attribute will result in a switch to a slower storage type.

```
const arr = [1, 2, 3];
// { elements: {
//   #0: 1,
//   #1: 2,
//   #2: 3
// }}

Object.defineProperty(arr, '0', { writable: false };
// { elements: {
//   #0: { value: 1, attrs: [_EC] },
//   #1: { value: 2, attrs: [WEC] },
//   #2: { value: 3, attrs: [WEC] }
// }}
```

My telegram channels:

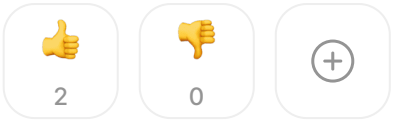
EN - https://t.me/frontend_almanac
RU - https://t.me/frontend_almanac_ru

Русская версия: https://blog.frontend-almanac.ru/js-object-structure

---

Roman Maksimov · March 31, 22:02 · 6.3K views · 2 reactions

Send tip One tipper Taras Protchenko



Leave a Comment