

03 | 快属性和慢属性：V8是怎样提升对象属性访问速度的？

2020-03-21 李兵

《图解 Google V8》

课程介绍 >



讲述：李兵

时长 13:22 大小 12.25M

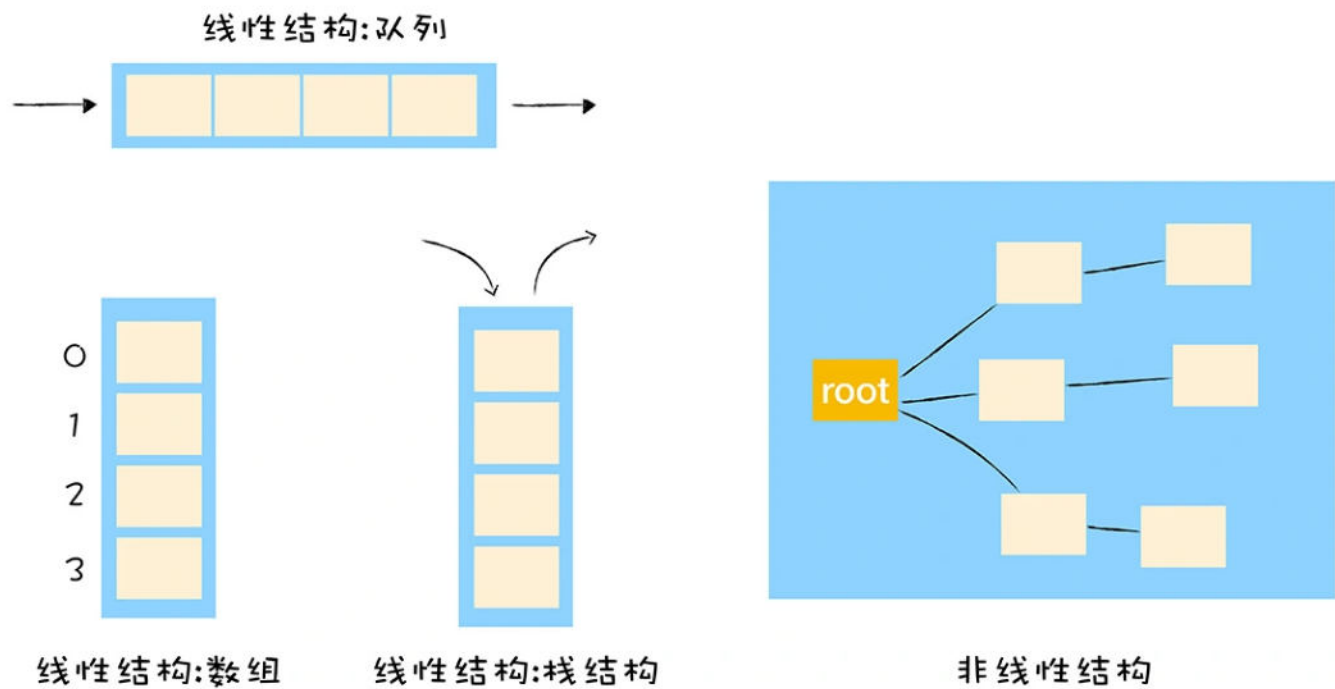


你好，我是李兵。

在前面的课程中，我们介绍了 JavaScript 中的对象是由一组组属性和值的集合，从 JavaScript 语言的角度来看，JavaScript 对象像一个字典，字符串作为键名，任意对象可以作为键值，可以通过键名读写键值。

然而在 V8 实现对象存储时，并没有完全采用字典的存储方式，这主要是出于性能的考量。因为字典是非线性的数据结构，查询效率会低于线性的数据结构，V8 为了提升存储和查找效率，采用了一套复杂的存储策略。





线性结构和非线性结构

今天这节课我们就来分析下 V8 采用了哪些策略提升了对象属性的访问速度。

常规属性 (properties) 和排序属性 (element)


在开始之前，我们先来了解什么是对象中的**常规属性**和**排序属性**，你可以先参考下面这样一段代码：

复制代码

```
1 function Foo() {
2     this[100] = 'test-100'
3     this[1] = 'test-1'
4     this["B"] = 'bar-B'
5     this[50] = 'test-50'
6     this[9] = 'test-9'
7     this[8] = 'test-8'
8     this[3] = 'test-3'
9     this[5] = 'test-5'
10    this["A"] = 'bar-A'
11    this["C"] = 'bar-C'
12 }
13 var bar = new Foo()
14
15
16 for(key in bar){
17     console.log(`index:${key} value:${bar[key]}`)
18 }
```



在上面这段代码中，我们利用构造函数 Foo 创建了一个 bar 对象，在构造函数中，我们给 bar 对象设置了很多属性，包括了数字属性和字符串属性，然后我们枚举出来了 bar 对象中所有的属性，并将其一一打印出来，下面就是执行这段代码所打印出来的结果：

 复制代码

```
1 index:1 value:test-1
2 index:3 value:test-3
3 index:5 value:test-5
4 index:8 value:test-8
5 index:9 value:test-9
6 index:50 value:test-50
7 index:100 value:test-100
8 index:B value:bar-B
9 index:A value:bar-A
10 index:C value:bar-C
```

观察这段打印出来的数据，我们发现打印出来的属性顺序并不是我们设置的顺序，我们设置属性的时候是乱序设置的，比如开始先设置 100，然后又设置了 1，但是输出的内容却非常规律，总的来说体现在以下两点：

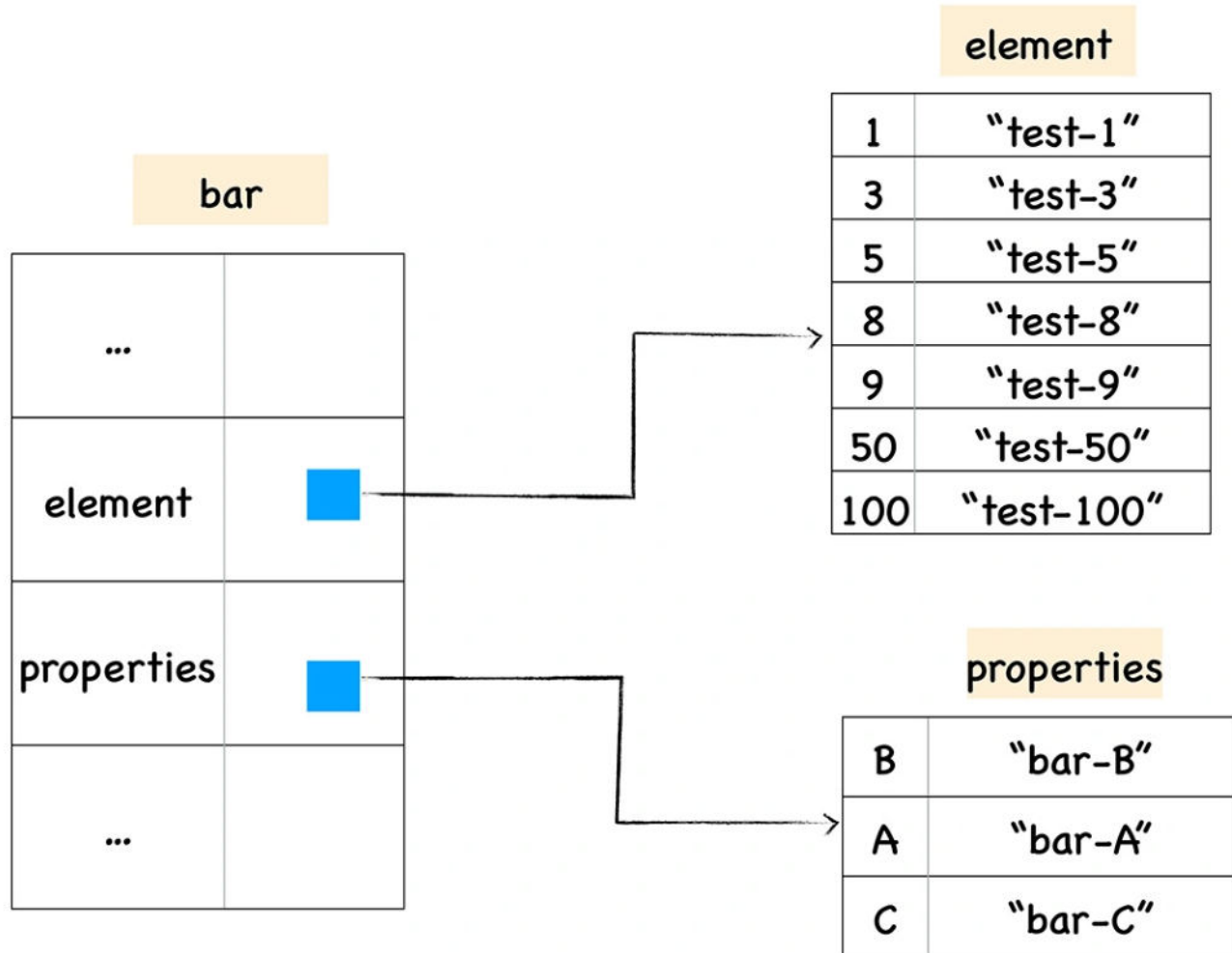
- 设置的数字属性被最先打印出来了，并且是按照数字大小的顺序打印的；
- 设置的字符串属性依然是按照之前的设置顺序打印的，比如我们是按照 B、A、C 的顺序设置的，打印出来依然是这个顺序。

之所以出现这样的结果，是因为在 ECMAScript 规范中定义了**数字属性应该按照索引值大小升序排列**，**字符串属性根据创建时的顺序升序排列**。

在这里我们把对象中的数字属性称为**排序属性**，在 V8 中被称为 **elements**，字符串属性就被称为**常规属性**，在 V8 中被称为 **properties**。

在 V8 内部，为了有效地提升存储和访问这两种属性的性能，分别使用了两个**线性数据结构**来分别保存排序属性和常规属性，具体结构如下图所示：





V8内部的对象构造

通过上图我们可以发现，bar 对象包含了两个隐藏属性：elements 属性和 properties 属性，elements 属性指向了 elements 对象，在 elements 对象中，会按照顺序存放排序属性，properties 属性则指向了 properties 对象，在 properties 对象中，会按照创建时的顺序保存了常规属性。

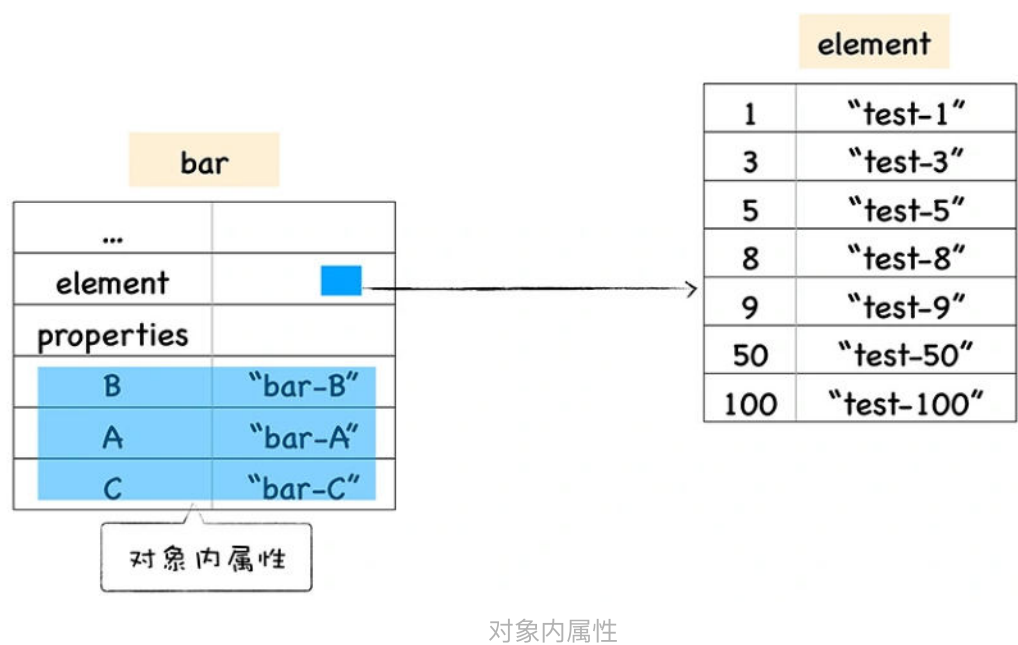
分解成这两种线性数据结构之后，如果执行索引操作，那么 V8 会先从 elements 属性中按照顺序读取所有的元素，然后再在 properties 属性中读取所有的元素，这样就完成一次索引操作。

快属性和慢属性

将不同的属性分别保存到 elements 属性和 properties 属性中，无疑简化了程序的复杂度，但是在查找元素时，却多了一步操作，比如执行 bar.B 这个语句来查找 B 的属性值，那么在 V8 会先查找出 properties 属性所指向的对象 properties，然后再在 properties 对象中查找 B 属性，这种方式在查找过程中增加了一步操作，因此会影响到元素的查找效率。



基于这个原因，V8 采取了一个权衡的策略以加快查找属性的效率，这个策略是将部分常规属性直接存储到对象本身，我们把这称为**对象内属性 (in-object properties)**。对象在内存中的展现形式你可以参看下图：



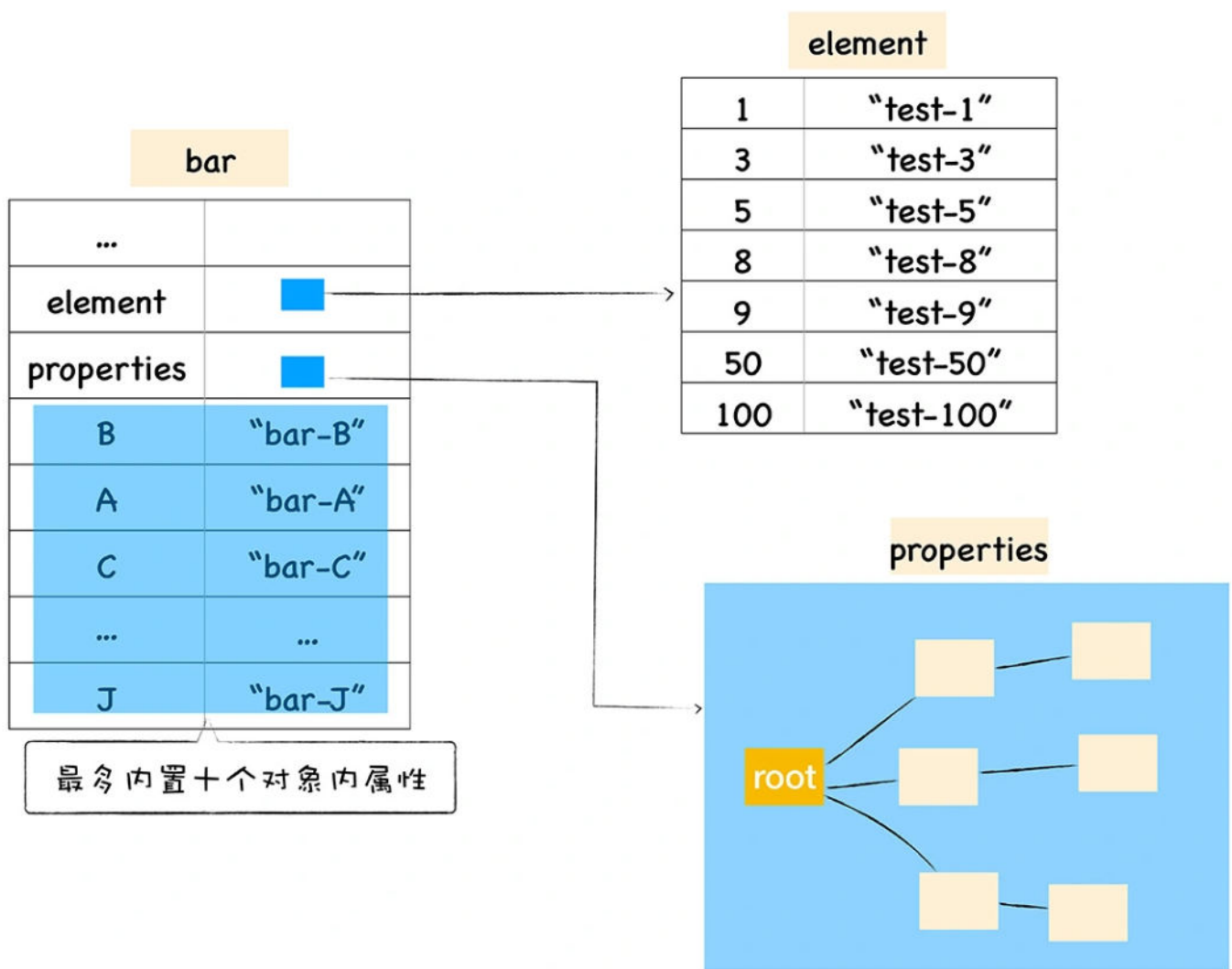
采用对象内属性之后，常规属性就被保存到 `bar` 对象本身了，这样当再次使用 `bar.B` 来查找 `B` 的属性值时，V8 就可以直接从 `bar` 对象本身去获取该值就可以了，这种方式减少查找属性值的步骤，增加了查找效率。

不过对象内属性的数量是固定的，默认是 10 个，如果添加的属性超出了对象分配的空间，则它们将被保存在常规属性存储中。虽然属性存储多了一层间接层，但可以自由地扩容。

通常，我们将保存在线性数据结构中的属性称之为“快属性”，因为线性数据结构中只需要通过索引即可以访问到属性，虽然访问线性结构的速度快，但是如果从线性结构中添加或者删除大量的属性时，则执行效率会非常低，这主要因为会产生大量时间和内存开销。

因此，如果一个对象的属性过多时，V8 就会采取另外一种存储策略，那就是“慢属性”策略，但慢属性的对象内部会有独立的非线性数据结构（词典）作为属性存储容器。所有的属性元信息不再是线性存储的，而是直接保存在属性字典中。





慢属性是如何存储的

实践：在 Chrome 中查看对象布局

现在我们知道了 V8 是怎么存储对象的了，接下来我们来结合 Chrome 中的内存快照，来看看对象在内存中是如何布局的？

你可以打开 Chrome 开发者工具，先选择控制台标签，然后在控制台中执行以下代码查看内存快照：

复制代码

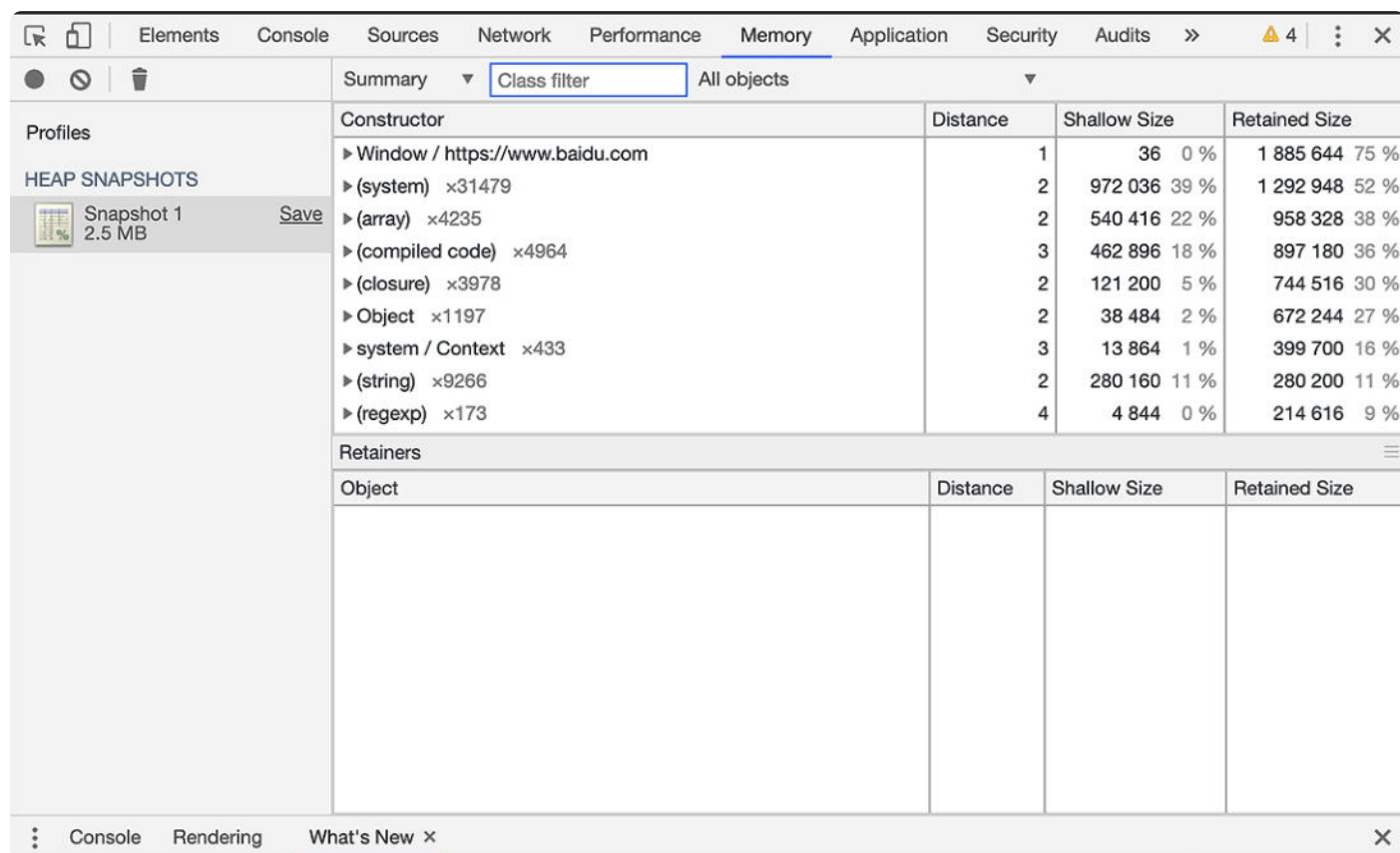
```
1 function Foo(property_num,element_num) {
2     //添加可索引属性
3     for (let i = 0; i < element_num; i++) {
4         this[i] = `element${i}`
5     }
6     //添加常规属性
7     for (let i = 0; i < property_num; i++) {
8         let ppt = `property${i}`
9         this[ppt] = ppt
10    }
```




```
11 }  
12 var bar = new Foo(10,10)
```

上面我们创建了一个构造函数，可以利用该构造函数创建新的对象，我给该构造函数设置了两个参数 `property_num`、`element_num`，分别代表创建常规属性的个数和排序属性的个数，我们先将这两种类型的个数都设置为 10 个，然后利用该构造函数创建了一个新的 `bar` 对象。

创建了函数对象，接下来我们就来看看构造函数和对象在内存中的状态。你可以将 Chrome 开发者工具切换到 Memory 标签，然后点击左侧的小圆圈就可以捕获当前的内存快照，最终截图如下所示：



Profiles	Summary	Class filter	All objects
HEAP SNAPSHOTS	Constructor		
Snapshot 1 2.5 MB	▶ Window / https://www.baidu.com		Distance: 1, Shallow Size: 36 0 %, Retained Size: 1 885 644 75 %
	▶ (system) x31479		Distance: 2, Shallow Size: 972 036 39 %, Retained Size: 1 292 948 52 %
	▶ (array) x4235		Distance: 2, Shallow Size: 540 416 22 %, Retained Size: 958 328 38 %
	▶ (compiled code) x4964		Distance: 3, Shallow Size: 462 896 18 %, Retained Size: 897 180 36 %
	▶ (closure) x3978		Distance: 2, Shallow Size: 121 200 5 %, Retained Size: 744 516 30 %
	▶ Object x1197		Distance: 2, Shallow Size: 38 484 2 %, Retained Size: 672 244 27 %
	▶ system / Context x433		Distance: 3, Shallow Size: 13 864 1 %, Retained Size: 399 700 16 %
	▶ (string) x9266		Distance: 2, Shallow Size: 280 160 11 %, Retained Size: 280 200 11 %
	▶ (regexp) x173		Distance: 4, Shallow Size: 4 844 0 %, Retained Size: 214 616 9 %
	Retainers		
	Object	Distance	Shallow Size, Retained Size

V8内存快照截图

上图就是收集了当前内存快照的界面，要想查找我们刚才创建的对象，你可以在搜索框里面输入构造函数 `Foo`，Chrome 会列出所有经过构造函数 `Foo` 创建的对象，如下图所示：



Summary ▼		Foo	All objects ▼
Constructor	Perspective	Distance	Shallow Size
▼ Foo		2	52 C
▼ Foo @31775 □		2	52 C
▶ map :: system / Map @101405		3	40 C
▶ __proto__ :: Object @93831 □		3	28 C
▶ elements :: (object elements)[] @101403		3	76 C
▶ property0 :: "property0" @93841 □		3	24 C
▶ property1 :: "property1" @93847 □		3	24 C
▶ property2 :: "property2" @93849 □		3	24 C
▶ property3 :: "property3" @93851 □		3	24 C
▶ property4 :: "property4" @93853 □		3	24 C
▶ property5 :: "property5" @93855 □		3	24 C
▶ property6 :: "property6" @93857 □		3	24 C
▶ property7 :: "property7" @93859 □		3	24 C
▶ property8 :: "property8" @93861 □		3	24 C
▶ property9 :: "property9" @93863 □		3	24 C

从内存快照搜索构造函数

观察上图，我们搜索出来了所有经过构造函数 Foo 创建的对象，点开 Foo 的那个下拉列表，第一个就是刚才创建的 bar 对象，我们可以看到 bar 对象有一个 elements 属性，这里面就包含我们创造的所有的排序属性，那么怎么没有常规属性对象呢？

这是因为只创建了 10 个常规属性，所以 V8 将这些常规属性直接做成了 bar 对象的对象内属性。

所以这时候的数据内存布局是这样的：

- 10 个常规属性作为对象内属性，存放在 bar 函数内部；
- 10 个排序属性存放在 elements 中。

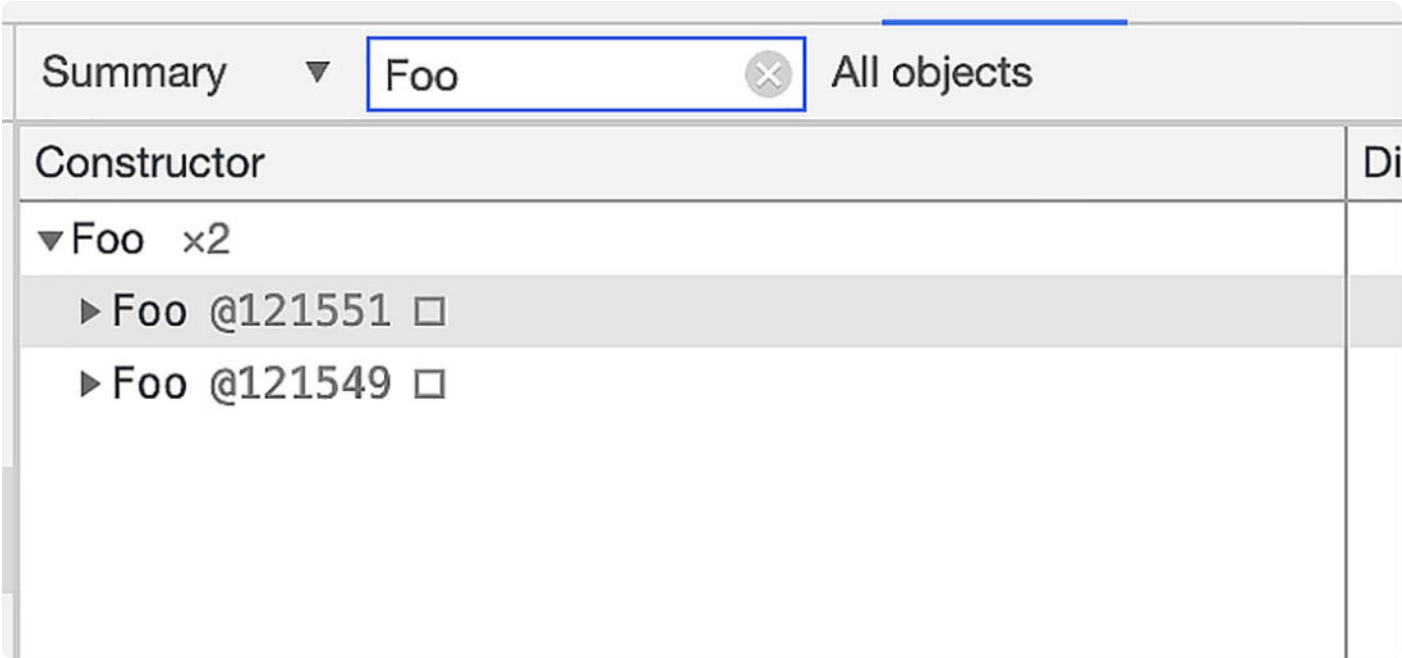
接下来我们可以将创建的对象属性的个数调整到 20 个，你可以在控制台执行下面这段代码：



```
1 var bar2 = new Foo(20,10)
```

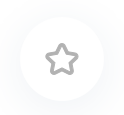
复制代码

然后我们再重新生成内存快照，再看看生成的图片：



利用构造函数生成的对象

我们可以看到，构造函数 Foo 下面已经有了两个对象了，其中一个 bar，另外一个为 bar2，我们点开第一个 bar2 对象，内容如下所示：



▼ Foo x2

▼ Foo @121551 □

```
▶ map :: system / Map @134565
▶ __proto__ :: Object @93831 □
▶ elements :: (object elements)[] @134563
▼ properties :: system @134561
  ▶ map :: system / Map @165
  ▶ 0 :: "property10" @134521 □
  ▶ 1 :: "property11" @134523 □
  ▶ 2 :: "property12" @134525 □
  ▶ 3 :: "property13" @134527 □
  ▶ 4 :: "property14" @134529 □
  ▶ 5 :: "property15" @134531 □
  ▶ 6 :: "property16" @134533 □
  ▶ 7 :: "property17" @134535 □
  ▶ 8 :: "property18" @134537 □
  ▶ 9 :: "property19" @134539 □
▶ property0 :: "property0" @93841 □
▶ property1 :: "property1" @93847 □
▶ property10 :: "property10" @134521 □
▶ property11 :: "property11" @134523 □
▶ property12 :: "property12" @134525 □
▶ property13 :: "property13" @134527 □
▶ property14 :: "property14" @134529 □
```

[查看对象属性](#)

由于创建的常用属性超过了 10 个，所以另外 10 个常用属性就被保存到 properties 中了，注意因为 properties 中只有 10 个属性，所以依然是线性的数据结构，我们可以看其都是按照创建时的顺序来排列的。

所以这时候属性的内存布局是这样的：

- 10 属性直接存放在 bar2 的对象内；
- 10 个常规属性以线性数据结构的方式存放在 properties 属性里面；
- 10 个数字属性存放在 elements 属性里面。



```
1 var bar3 = new Foo(100,10)
```

Summary ▾ Foo × All objects ▾

Constructor


▼ Foo x3

▼ Foo @138959 □

▼ properties :: (object properties)[] @140759

- ▶ map :: system / Map @149
- ▶ 101 :: "property30" @139249 □
- ▶ 102 :: "property30" @139249 □
- ▶ 104 :: "property47" @139307 □
- ▶ 105 :: "property47" @139307 □
- ▶ 107 :: "property25" @139415 □
- ▶ 108 :: "property25" @139415 □
- ▶ 11 :: "property8" @93861 □
- ▶ 110 :: "property51" @139265 □
- ▶ 111 :: "property51" @139265 □
- ▶ 113 :: "property14" @134529 □
- ▶ 114 :: "property14" @134529 □
- ▶ 119 :: "property54" @139283 □
- ▶ 12 :: "property8" @93861 □
- ▶ 120 :: "property54" @139283 □
- ▶ 122 :: "property97" @139413 □
- ▶ 123 :: "property97" @139413 □
- ▶ 128 :: "property26" @139303 □
- ▶ 129 :: "property26" @139303 □
- ▶ 143 :: "property64" @139227 □
- ▶ 144 :: "property64" @139227 □

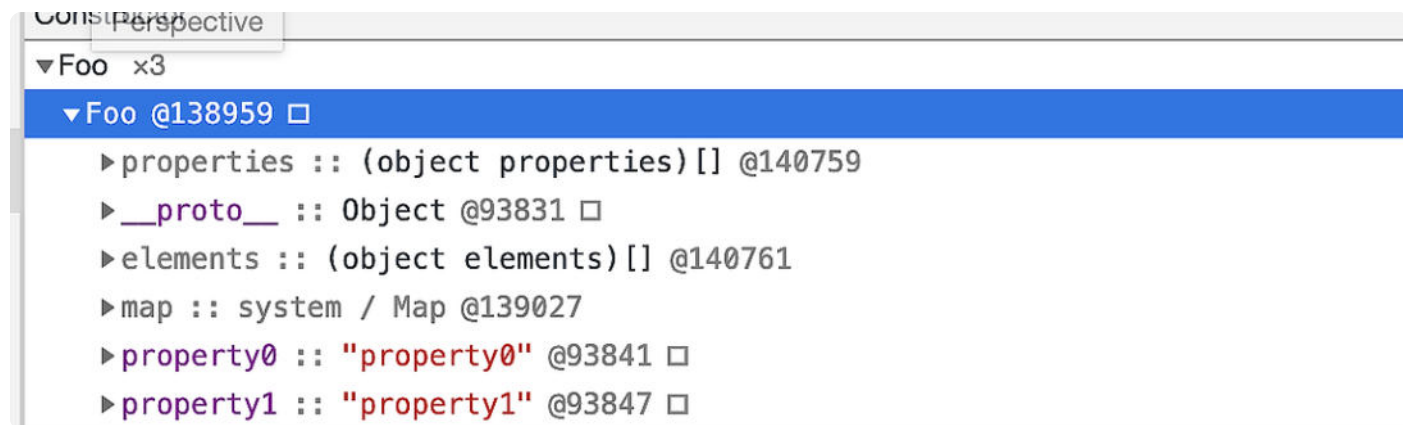
结合上图，我们可以看到，这时候的 `properties` 属性里面的数据并不是线性存储的，而是以非线性的字典形式存储的，所以这时候属性的内存布局是这样的：

- 

- 10 个数字属性存放在 elements 属性里面。

其他属性

好了，现在我们知道 V8 是怎么存储对象的了，不过这里还有几个重要的隐藏属性我还没有介绍，下面我们就来简单地看下。你可以先看下图：



其他隐藏属性

观察上图，除了 elements 和 properties 属性，V8 还为每个对象实现了 map 属性和 __proto__ 属性。__proto__ 属性就是原型，是用来实现 JavaScript 继承的，我们会在下一节来介绍；而 map 则是隐藏类，我们会在《[15 | 隐藏类：如何在内存中快速查找对象属性？](#)》这一节中介绍其工作机制。

总结

好了，本节的内容就介绍到这里，下面我来总结下本文的主要内容：

本文我们的主要目标是介绍 V8 内部是如何存储对象的，因为 JavaScript 中的对象是由一组组属性和值组成的，所以最简单的方式是使用一个字典来保存属性和值，但是由于字典是非线性结构，所以如果使用字典，读取效率会大大降低。

为了提升查找效率，V8 在对象中添加了两个隐藏属性，排序属性和常规属性，element 属性指向了 elements 对象，在 elements 对象中，会按照顺序存放排序属性。properties 属性则指向了 properties 对象，在 properties 对象中，会按照创建时的顺序保存常规属性。

通过引入这两个属性，加速了 V8 查找属性的速度，为了更加进一步提升查找效率，V8 还实现了内置内属性的策略，当常规属性少于一定数量时，V8 就会将这些常规属性直接写进对象中，这样又节省了一个中间步骤。



但是如果对象中的属性过多时，或者存在反复添加或者删除属性的操作，那么 V8 就会将线性的存储模式降级为非线性的字典存储模式，这样虽然降低了查找速度，但是却提升了修改对象的属性的速度。


思考题

通常，我们不建议使用 delete 来删除属性，你能结合文中介绍的快属性和慢属性，给出不建议使用 delete 的原因吗？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

 生成海报并分享

 赞 14  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 函数即对象：一篇文章彻底搞懂JavaScript的函数特点

下一篇 04 | 函数表达式：涉及大量概念，函数表达式到底该怎么学？



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (58)

写留言



伏枫

2020-03-23

<https://www.cnblogs.com/chargeworld/p/12236848.html>

找到了一篇博客，应该能帮助一些同学解惑

作者回复: 很赞



44



neohope

2020-07-21

1、chrome显示

不要关心一级目录上是否存在某个element或property，为了调试方便，chrome应该是无论如何存储，都会输出来。

直接去看elements和properties内存储的内容，更准确一些。

2、截图里property10怎么有两个：

这个问题，建议最好改一下演示代码，将Key和Value区分开，现在两个一样，容易引起一些误解。

3、element



element没有内置。

element默认应该采用连续的存储结构，通过浪费空间换取时间，直接下标访问，提升访问速度。

但当element的序号十分不连续时，会优化成为hash表，因为要浪费的空间太大了，不合算。

4、property

property有内置，只有十个，但建议把这十个单独考虑，后面就容易考虑清楚了。

property默认采用链表结构，当数据量很小时，查找也会很快，但数据量上升到某个数值后，会优化成为hash表。

因为超过某个数值，顺序查找就不够快了，需要通过hash表结构查找，提升速度。

5、hash表不是应该查找一次吗？为何是慢查询

hash表要解决key冲突问题，一般会用list存储多个冲突的key，所以计算hash后，还是要做顺序访问，所以要多次访问。

此外，还涉及到hash扩容的问题，那就更慢了。

所以，整体上来说，hash慢于按地址访问的；

在数据量小的时候，也慢于链表的顺序访问。

6、hash表如何存储property顺序？

再用一个链表记录插入属性就好了，类似于Java中的 LinkedHashMap，就可以解决问题



39



cc

2020-03-22

有个疑问，properties在元素较少的时候使用链表存储的吗？在元素较多的时候换成查找树？properties存的属性key是字符串，应该不可能是数组存。要不就是链表，要不就是hash表。如果是hash表，那就没有必要切换成查找树，性能改变微乎其微，最多也就是把hash表里由于冲突导致的过长链表换成查找树。

对文章里所说的非线性结构和线性结构感到很困惑，比如链表和数组的查找性能就有很大区别，但又都是线性结构。所以为啥不直接说具体是数组还是链表？

字典的实现可以是哈希表或者查找树，哈希表是线性结构，查找树是非线性结构。

这节看下来这真是一头雾水。

共 8 条评论 >

20



潇潇雨歇

2020-03-22

使用delete删除属性：

如果删除属性在线性结构中，删除后需要移动元素，开销较大，而且可能需要将慢属性重排到快属性。

如果删除属性在properties对象中，查找开销较大。

共 5 条评论 >

👍 16



try-catch

2020-03-24

执行完例子后有些疑惑，找到了v8引擎原博客 <https://v8.dev/blog/fast-properties> 中找到了答案：

"The number of in-object properties is predetermined by the initial size of the object"
in-object properties size 取决于初始化对象的大小。

作者回复：赞，v8.dev里面的文章都不错



👍 14



Silence

2020-03-21

老师，我的 Chrome 版本是 80 的，看 memory 面板好像和你讲的不太一样。

当有 20 个常规属性时，properties 中有 10 个，但是20个都在 bar 对象内。

当有 100 个常规属性时，properties 就更诡异了，每个都有 2 个，共 200 个，bar 对象上有 100 个。

而且每次都是刷新浏览器后试的，这是什么情况？

评论区没办法截图。

共 9 条评论 >

👍 11



王楚然

2020-03-23

有几个问题没有弄懂：

1. element（排序属性）是否也有内置，快属性，慢属性三种？不会是一直线性存储吧？
2. 在properties（字符串属性）很多的时候，会大部分存储成字典结构，具体是什么样的字典结构呢？如何按照ECMA标准保证属性依据创建顺序排序呢？
3. 还有针对原文“线性的存储模式降级为非线性的字典存储模式，这样虽然降低了查找速度，但是却提升了修改对象的属性的速度。”这句话，线性存储模式是链表吗？字典存储是什么呢？修改的流程，应该也是先查找后修改吧？为什么后者会降低查找速度却能提高修改速度呢？



👍 7



青史成灰

老师，这里的线形、非线性数据结构，能否说的具体点，是数组，链表，红黑树还是啥的



5



拼命的小贝壳

2020-03-21

这个快属性的数量和平台相关么？ mac 平台 chrome 尝试这个代码，会把所有 properties elements 添加为快属性。

不建议 delete 可能会影响性能的地方：

- 1.如果删除排序属性，线性存储结构会有个 $O(n)$ 复杂度的移动。
- 2.如果删除常规属性，可能会重新计算并添加快属性。



4



一步

2020-03-21

关于常规属性过多时候的表现我这里有2个问题想请教一下：

- 1、我这里和老师实验结果不一样：我这里利用 Chrome 创建了 30个常规属性，我看了一下是没有使用对象内属性的，30 个属性以字典的形式保存的 properties 属性对象中
 - 2、当转化为字典后，properties 对象是怎么生成的，每个属性的值为什么会出现2次，那个属性的值的 key 是怎么生成的
- (判断属性是否过多是以 25为界限的)



3



马成

2020-03-21

老师，字典结构为什么读取效率比线性结构低。如果都是数字索引的话，线性结构很快，但是字符串属性只能遍历呀，怎么会比字典快呢？

共 3 条评论 >

3



dellyoung

2020-04-04

排查内存泄露也需要用到，Memory，李大大能后边补充一节如何排查内存泄露吗，感觉挺常用，面试中也经常被问到，感谢！！

共 1 条评论 >

2



宇宙全栈

2020-03-21

在 Chrome 开发者工具中实践时，发现了一个问题：在当前这个页面，打开开发者工具，在 Console 中执行代码后，在 Memory 中生成快照，但是在快照中未找到 Foo，并且快照只有



761KB。此时，在保持开发者工具打开的状态下刷新页面，在 Memeory 中再次生成快照，这时在快照中找到了 Foo，并且快照有 77.1MB。老师能否解释下这个现象😄



👍 2



Condor Hero

2021-09-06

看完快属性慢属性、隐藏类、内联缓存，对对象是非常的了解了，但是对 ES6 的 Map 和 Set 疑问就来了，它们很快，但是为什么快呢。

希望老师加餐😋



👍 1



HoSalt

2020-04-24

词典和字典是怎样的数据结构，类似于树？

作者回复: 就是hash表

共 2 条评论 >

👍 1



Lorin

2020-03-25

老师，hash表和js中对象是什么关系？我感觉对象就是哈希表，但是我看哈希表的定义里面key会经过哈希函数进行编码，这之间有什么区别呢？

作者回复: 你可以把对象看成是一个hash表，但是V8为了性能，做了很多改进



👍 1



一步

2020-03-21

老师有个疑问就是：当常规属性的数量大于对象内属性的数量10限制后，就会按照创建顺序先把常规属性放到对象内属性中，然后再把剩余的常规属性放到 properties 属性中，这样的话。当我查询一个常规属性的时候，就需要查询两次：先查询对象内属性，没有的话在查询 properties 属性。

这里多查询了一次，为什么不这样设计呢？当常规属性的数量大于对象内属性的数量10限制后就不使用对象内属性了，直接使用properties 属性，这样不就会减少一次查询吗？



👍 1



一步 

2020-03-21

字典存储模式，这样虽然降低了查找速度，但是却提升了修改对象的属性的速度。

字段存储为了降低了查找效率呢？字典不是 $O(1)$ ，直接就可以索引到的，是因为字段的 key 的 hash 有可能冲突吗？然后退化成链表？



1



Hyhang

2022-01-20

inobject properties 的数量是有上限的，其计算过程大致是：

// 为了方便计算，这里把涉及到的常量定义从源码各个文件中摘出后放到了一起

```
#if V8_HOST_ARCH_64_BIT
```

```
constexpr int kSystemPointerSizeLog2 = 3;
```

```
#endif
```

```
constexpr int kTaggedSizeLog2 = kSystemPointerSizeLog2;
```

```
constexpr int kSystemPointerSize = sizeof(void*);
```

```
static const int kJSObjectHeaderSize = 3 * kApiTaggedSize;
```

```
STATIC_ASSERT(kHeaderSize == Internals::kJSObjectHeaderSize);
```

```
constexpr int kTaggedSize = kSystemPointerSize;
```

```
static const int kMaxInstanceSize = 255 * kTaggedSize;
```

```
static const int kMaxInObjectProperties = (kMaxInstanceSize - kHeaderSize) >> kTaggedSizeLog2;
```

根据上面的定义，在 64bit 系统上、未开启指针压缩的情况下，最大数量是 $252 = (255 * 8 - 3 * 8) / 8$



z

2022-01-03

老师，我按照我的理解测试了一下，最终结果是直接获取对象的 key 比 map 快呀？是我理解的不对么？下面是代码：

```
function getRandomStr(len){  
    return Math.random().toString(36).slice(2, len)  
}  
  
let record = {}  
let key = ""  
for(let i = 0; i < 100000; i++) {
```



```
    if(i === 5555) {  
        key = getRandomStr(11)  
        record[key] = i;  
    } else {  
        record[getRandomStr(11)] = i;  
    }  
}  
console.time("对象获取速度")  
record[key]  
console.timeEnd("对象获取速度")
```

```
console.time("r2m")  
let map = new Map(Object.entries(record))  
console.timeEnd("r2m")
```

```
console.time("map获取速度")  
map.get(key)  
console.timeEnd("map获取速度")
```

