

## 15 | 隐藏类：如何在内存中快速查找对象属性？

2020-04-18 李兵

图解 Google V8

[进入课程 >](#)



讲述：李兵

时长 17:41 大小 16.20M



你好，我是李兵。

我们知道 JavaScript 是一门动态语言，其执行效率要低于静态语言，V8 为了提升 JavaScript 的执行速度，借鉴了很多静态语言的特性，比如实现了 JIT 机制，为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。

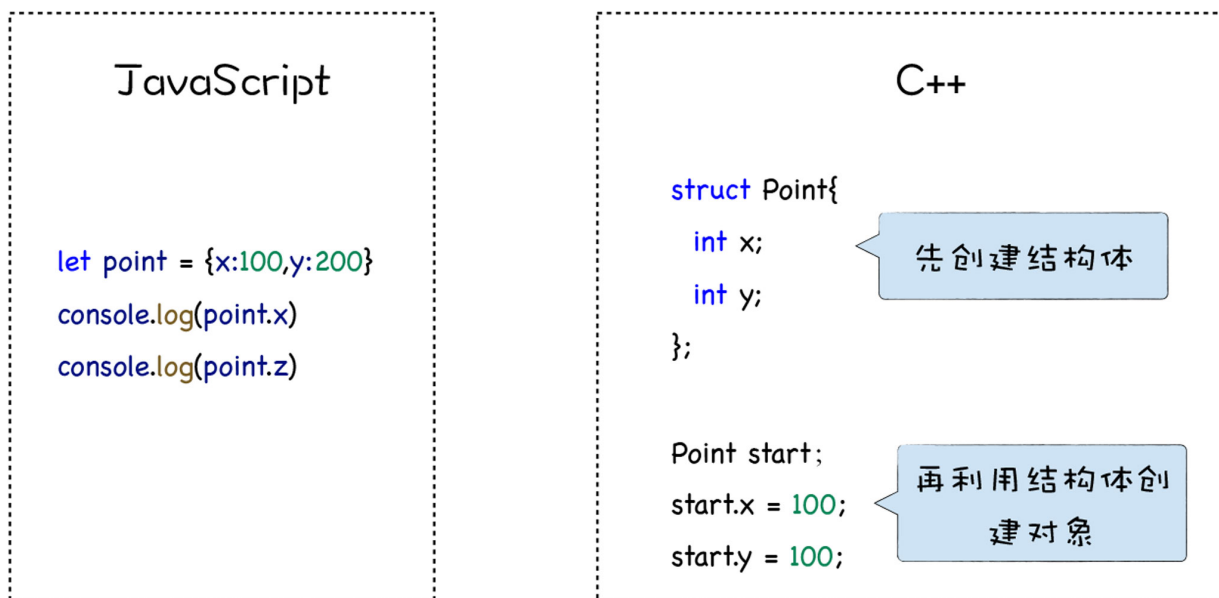
今天我们来重点分析下 V8 中的隐藏类，看看它是怎么提升访问对象属性值速度的。



为什么静态语言的效率更高？

由于隐藏类借鉴了部分静态语言的特性，因此要解释清楚这个问题，我们就先来分析下为什么静态语言比动态语言的执行效率更高。

我们通过下面两段代码，来对比一下动态语言和静态语言在运行时的一些特征，一段是动态语言的 JavaScript，另外一段静态语言的 C++ 的源码，具体源码你可以参看下图：



那么在运行时，这两段代码的执行过程有什么区别呢？

我们知道，JavaScript 在运行时，对象的属性是可以被修改的，所以当 V8 使用了一个对象时，比如使用了 `start.x` 的时候，它并不知道该对象中是否有 `x`，也不知道 `x` 相对于对象的偏移量是多少，也可以说 V8 并不知道该对象的具体形状。

那么，当在 JavaScript 中要查询对象 `start` 中的 `x` 属性时，V8 会按照具体的规则一步一步来查询，这个过程非常的慢且耗时（具体查找过程你可以参考《[03 | 快属性和慢属性：V8 采用了哪些策略提升了对象属性的访问速度？](#)》这节课程中的内容）。

这种动态查询对象属性的方式和 C++ 这种静态语言不同，C++ 在声明一个对象之前需要定义该对象的结构，我们也可以称为形状，比如 `Point` 结构体就是一种形状，我们可以使用这个形状来定义具体的对象。

C++ 代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中，`Point` 的形状是无法被改变的。

那么在 C++ 中访问一个对象的属性时，自然就知道该属性相对于该对象地址的偏移值了，比如在 C++ 中使用 start.x 的时候，编译器会直接将 x 相对于 start 的地址写进汇编指令中，那么当使用了对象 start 中的 x 属性时，CPU 就可以直接去内存地址中取出该内容即可，没有任何中间的查找环节。

因为静态语言中，可以直接通过偏移量查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

## 什么是隐藏类 (Hidden Class) ?

既然静态语言的查询效率这么高，那么是否能将这种静态的特性引入到 V8 中呢？

答案是可行的。

目前所采用的一个思路就是将 JavaScript 中的对象静态化，也就是 V8 在运行 JavaScript 的过程中，会假设 JavaScript 中的对象是静态的，具体地讲，V8 对每个对象做如下两点假设：

对象创建好了之后就不会添加新的属性；

对象创建好了之后也不会删除属性。

符合这两个假设之后，V8 就可以对 JavaScript 中的对象做深度优化了，那么怎么优化呢？

具体地讲，V8 会为每个对象创建一个隐藏类，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

对象中所包含的所有的属性；

每种类型相对于对象的偏移量。

有了隐藏类之后，那么当 V8 访问某个对象中的某个属性时，就会先去隐藏类中查找该属性相对于它的对象的偏移量，有了偏移量和属性类型，V8 就可以直接去内存中取出对于的属性值，而不需要经历一系列的查找过程，那么这就大大提升了 V8 查找对象的效率。

我们可以结合一段代码来分析下隐藏类是怎么工作的：

```
1 let point = {x:100,y:200}
```

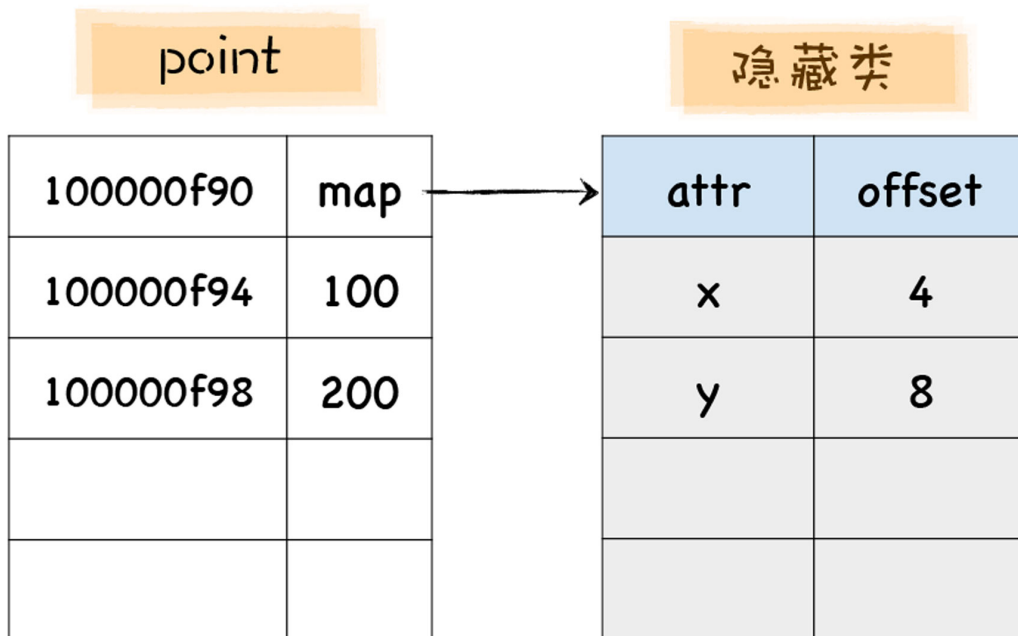
当 V8 执行到这段代码时，会先为 point 对象创建一个隐藏类，在 V8 中，把隐藏类又称为 **map**，每个对象都有一个 map 属性，其值指向内存中的隐藏类。

隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的偏移量，比如 point 对象的隐藏类就包括了 x 和 y 属性，x 的偏移量是 4，y 的偏移量是 8。

### point 对象的隐藏类

attr	offset
x	4
y	8

注意，这是 point 对象的 map，它不是 point 对象本身。关于 point 对象和 map 之间的关系，你可以参看下图：



在这张图中，左边的是 point 对象在内存中的布局，右边是 point 对象的 map，我们可以看到，point 对象的第一个属性就指向了它的 map，关于如何通过浏览器查看对象的 map，我们在《03 | 快属性和慢属性：V8 采用了哪些策略提升了对象属性的访问速度？》这节课也做过简单的分析，你可以回顾下这节内容。

有了 map 之后，当你再次使用 point.x 访问 x 属性时，V8 会查询 point 的 map 中 x 属性相对 point 对象的偏移量，然后将 point 对象的起始位置加上偏移量，就得到了 x 属性的值在内存中的位置，有了这个位置也就拿到了 x 的值，这样我们就省去了一个比较复杂的查找过程。

这就是将动态语言静态化的一个操作，V8 通过引入隐藏类，模拟 C++ 这种静态语言的机制，从而达到静态语言的执行效率。

## 实践：通过 d8 查看隐藏类

了解了隐藏类的工作机制，我们可以使用 d8 提供的 API DebugPrint 来查看 point 对象中的隐藏类。

复制代码


```
1 let point = {x:100,y:200};  
2 %DebugPrint(point);
```



这里你需要注意，在使用 d8 内部 API 时，有一点很容易出错，就是需要为 JavaScript 代码加上分号，不然 d8 会报错，所以这段代码里面我都加上了分号。

然后将下面这段代码保存 test.js 文件中，再执行：

```
1 d8 --allow-natives-syntax test.js
```

 复制代码

执行这段命令，就可以打印出 point 对象的基础结构了，打印出来的结果如下所示：

```
1 DebugPrint: 0x19dc080c5af5: [JS_OBJECT_TYPE]
2   - map: 0x19dc08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
3   - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
4   - elements: 0x19dc080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
5   - properties: 0x19dc080406e9 <FixedArray[0]> {
6     #x: 100 (const data field 0)
7     #y: 200 (const data field 1)
8   }
9 0x19dc08284d11: [Map]
10   - type: JS_OBJECT_TYPE
11   - instance size: 20
12   - inobject properties: 2
13   - elements kind: HOLEY_ELEMENTS
14   - unused property fields: 0
15   - enum length: invalid
16   - stable_map
17   - back pointer: 0x19dc08284ce9 <Map(HOLEY_ELEMENTS)>
18   - prototype_validity cell: 0x19dc081c0451 <Cell value= 1>
19   - instance descriptors (own) #2: 0x19dc080c5b25 <DescriptorArray[2]>
20   - prototype: 0x19dc08241151 <Object map = 0x19dc082801c1>
21   - constructor: 0x19dc0824116d <JSFunction Object (sfi = 0x19dc081c55ad)>
22   - dependent code: 0x19dc080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
23   - construction counter: 0
```

 复制代码

从这段 point 的内存结构中，我们可以看到，point 对象的第一个属性就是 map，它指向了 0x19dc08284d11 这个地址，这个地址就是 V8 为 point 对象创建的隐藏类，除了 map 属性之外，还有我们之前介绍过的 prototype 属性，elements 属性和 properties 属性（关于这些属性的函数，你可以参看《[03 | 快属性和慢属性：V8 采用了哪些策略提升](#)

了对象属性的访问速度?》和《05 | 原型链: V8 是如何实现对象继承的?》这两节的内容)。

## 多个对象共用一个隐藏类

现在我们知道了在 V8 中, 每个对象都有一个 map 属性, 该属性值指向该对象的隐藏类。

不过如果两个对象的形状是相同的, V8 就会为其复用同一个隐藏类, 这样有两个好处:

1. 减少隐藏类的创建次数, 也间接加速了代码的执行速度;
2. 减少了隐藏类的存储空间。

那么, 什么情况下两个对象的形状是相同的, 要满足以下两点:

相同的属性名称;

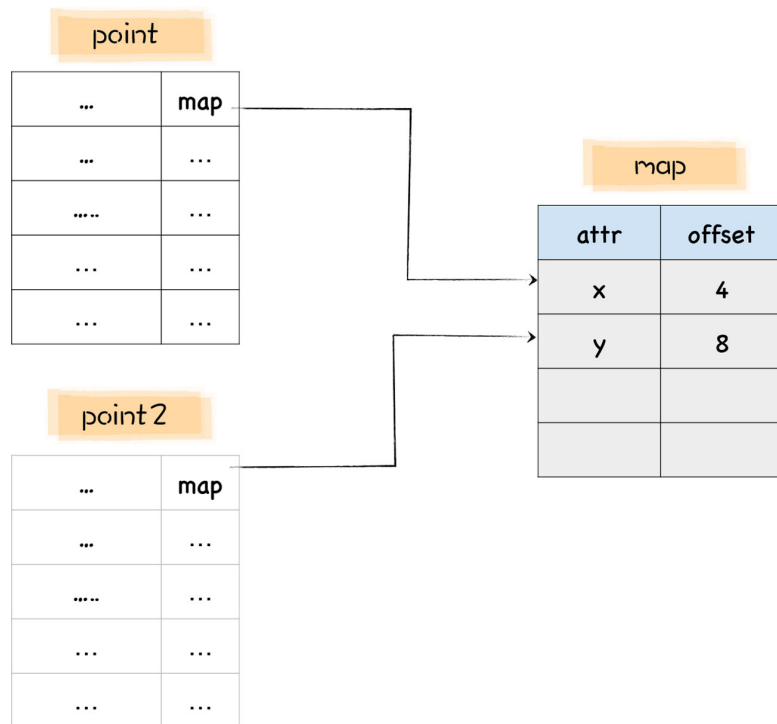
相等的属性个数。

接下来我们就来创建两个形状一样的对象, 然后看看它们的 map 属性是不是指向了同一个隐藏类, 你可以参看下面的代码:

```
1 let point = {x:100,y:200};
2 let point2 = {x:3,y:4};
3 %DebugPrint(point);
4 %DebugPrint(point2);
```

复制代码

当 V8 执行到这段代码时, 首先会为 point 对象创建一个隐藏类, 然后继续创建 point2 对象。在创建 point2 对象的过程中, 发现它的形状和 point 是一样的。这时候, V8 就会将 point 的隐藏类给 point2 复用, 具体效果你可以参看下图:



你也可以使用 d8 来证实下，同样使用这个命令：

```
1 d8 --allow-natives-syntax test.js
```

[复制代码](#)

打印出来的 point 和 point2 对象，你会发现它们的 map 属性都指向了同一个地址，这也就意味着它们共用了同一个 map。

## 重新构建隐藏类

关于隐藏类，还有一个问题你需要注意一下。在这节课开头我们提到了，V8 为了实现隐藏类，需要两个假设条件：

对象创建好了之后就不会添加新的属性；

对象创建好了之后也不会删除属性。


但是，JavaScript 依然是动态语言，在执行过程中，对象的形状是可以被改变的，如果某个对象的形状改变了，隐藏类也会随着改变，这意味着 V8 要为新改变的对象重新构建新的隐藏类，这对于 V8 的执行效率来说，是一笔大的开销。



通俗地理解，**给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状**，那么势必也就会触发 V8 为改变形状后的对象重建新的隐藏类。

我们可以看一个简单的例子：

```
1 let point = {};  
2 %DebugPrint(point);  
3 point.x = 100;  
4 %DebugPrint(point);  
5 point.y = 200;  
6 %DebugPrint(point);
```

 复制代码

将这段代码保存到 test.js 文件中，然后执行：

```
1 d8 --allow-natives-syntax test.js
```

 复制代码

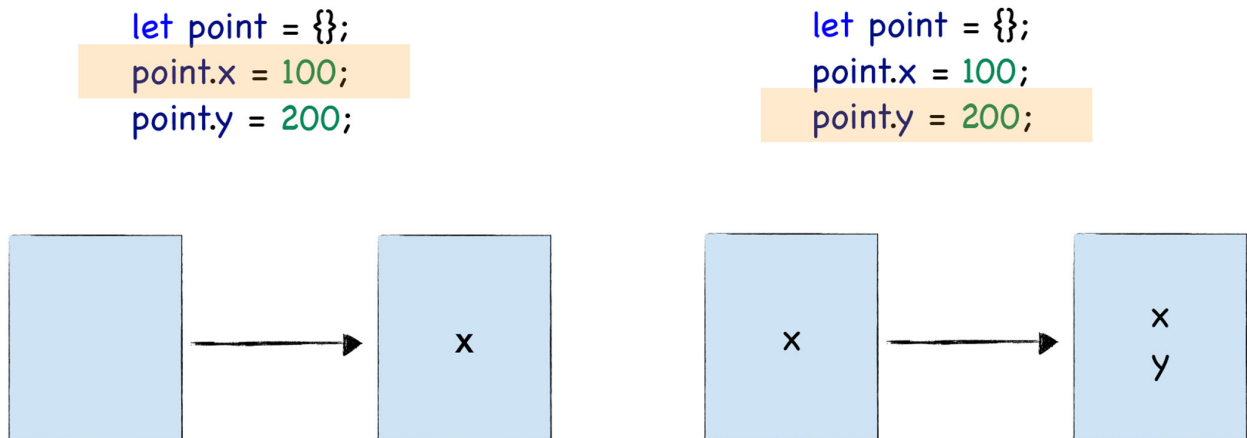
执行这段命令，d8 会打印出来不同阶段的 point 对象所指向的隐藏类，在这里我们只关心 point 对象 map 的指向，所以我将其他的一些信息都省略了，最终打印出来的结果如下所示：

```
1 DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]  
2 - map: 0x0986082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]  
3 - ...  
4  
5  
6 DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]  
7 - map: 0x098608284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]  
8 - ...  
9 - properties: 0x0986080406e9 <FixedArray[0]> {  
10   #x: 100 (const data field 0)  
11 }  
12  
13  
14 DebugPrint: 0x986080c5b35: [JS_OBJECT_TYPE]  
15 - map: 0x098608284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]  
16 - p  
17 - ...  
18 - properties: 0x0986080406e9 <FixedArray[0]> {
```

 复制代码

```
19      #x: 100 (const data field 0)
20      #y: 200 (const data field 1)
```

根据这个打印出来的结果，我们可以明显看到，每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了，改变过程你可以参看下图：



同样，如果你删除了对象的某个属性，那么对象的形状也就随着发生了改变，这时 V8 也会重建该对象的隐藏类，我们可以看下面这样的一个例子：

```
1 let point = {x:100,y:200};
2 delete point.x
```

复制代码

我们再次使用 d8 来打印这段代码中不同阶段的 point 对象属性，移除多余的信息，最终打印出来的结果如下所示

```
1 DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
2 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
3 -...
4 - properties: 0x1c2f080406e9 <FixedArray[0]> {
5     #x: 100 (const data field 0)
6     #y: 200 (const data field 1)
7 }
8
9
10 DebugPrint: 0x1c2f080c5b1d: [JS_OBJECT_TYPE]
11 - map: 0x1c2f08284d11 <Map(HOLEY_ELEMENTS)> [FastProperties]
```

复制代码

```
12 - ...
13 - properties: 0x1c2f08045567 <FixedArray[0]> {
14   #y: 200 (const data field 1)
15 }
```

## 最佳实践

好了，现在我们知道了 V8 会为每个对象分配一个隐藏类，在执行过程中：

如果对象的形状没有发生改变，那么该对象就会一直使用该隐藏类；

如果对象的形状发生了改变，那么 V8 会重建一个新的隐藏类给该对象。

我们当然希望对象中的隐藏类不要随便被改变，因为这样会触发 V8 重构该对象的隐藏类，直接影响到了程序的执行性能。那么在实际工作中，我们应该尽量注意以下几点：

**一，使用字面量初始化对象时，要保证属性的顺序是一致的。**比如先通过字面量 x、y 的顺序创建了一个 point 对象，然后通过字面量 y、x 的顺序创建一个对象 point2，代码如下所示：

```
1 let point = {x:100,y:200};
2 let point2 = {y:100,x:200};
```

 复制代码

虽然创建时的对象属性一样，但是它们初始化的顺序不一样，这也会导致形状不同，所以它们会有不同的隐藏类，所以我们要尽量避免这种情况。

**二，尽量使用字面量一次性初始化完整对象属性。**因为每次为对象添加一个属性时，V8 都会为该对象重新设置隐藏类。

**三，尽量避免使用 delete 方法。**delete 方法会破坏对象的形状，同样会导致 V8 为该对象重新生成新的隐藏类。

## 总结

这节课我们介绍了 V8 中隐藏类的工作机制，我们先分析了 V8 引入隐藏类的动机。因为 JavaScript 是一门动态语言，对象属性在执行过程中是可以被修改的，这就导致了在运行时，V8 无法知道对象的完整形状，那么当查找对象中的属性时，V8 就需要经过一系列复杂的步骤才能获取到对象属性。

为了加速查找对象属性的速度，V8 在背后为每个对象提供了一个隐藏类，隐藏类描述了该对象的具体形状。有了隐藏类，V8 就可以根据隐藏类中描述的偏移地址获取对应的属性值，这样就省去了复杂的查找流程。

不过隐藏类是建立在两个假设基础之上的：

对象创建好了之后就不会添加新的属性；

对象创建好了之后也不会删除属性。

一旦对象的形状发生了改变，这意味着 V8 需要为对象重建新的隐藏类，这就会带来效率问题。为了避免一些不必要的性能问题，我们在程序中尽量不要随意改变对象的形状。我在这节课中也给你列举了几个最佳实践的策略。

最后，关于隐藏类，我们记住以下几点。

在 V8 中，每个对象都有一个隐藏类，隐藏类在 V8 中又被称为 map。

在 V8 中，每个对象的第一个属性的指针都指向其 map 地址。

map 描述了其对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少？

如果添加新的属性，那么需要重新构建隐藏类。

如果删除了对象中的某个属性，通用也需要构建隐藏类。

## 思考题

现在我们知道了 V8 为每个对象配置了一个隐藏类，隐藏类描述了该对象的形状，V8 可以通过隐藏类快速获取对象的属性值。不过这里还有另外一类问题需要考虑。

比如我定义了一个获取对象属性值的函数 loadX，loadX 有一个参数，然后返回该参数的 x 属性值：

 复制代码

```
1 function loadX(o) {  
2     return o.x  
3 }  
4 var o = { x: 1,y:3}  
5 var o1 = { x: 3 ,y:6}  
6 for (var i = 0; i < 90000; i++) {  
7     loadX(o)  
8     loadX(o1)  
9 }
```

当 V8 调用 loadX 的时候，会先查找参数 o 的隐藏类，然后利用隐藏类中的 x 属性的偏移量查找到 x 的属性值，虽然利用隐藏类能够快速提升对象属性的查找速度，但是依然有一个查找隐藏类和查找隐藏类中的偏移量两个操作，如果 loadX 在代码中会被重复执行，依然影响到了属性的查找效率。

那么留给你的问题是：如果你是 V8 的设计者，你会采用什么措施来提高 loadX 函数的执行效率？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。

# 图解 Google V8

一门课搞懂 JavaScript 执行逻辑

李兵

前盛大创新院高级研究员



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 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 14 | 字节码（二）：解释器是如何解释执行字节码的？

下一篇 16 | 答疑：V8是怎么通过内联缓存来提升函数执行效率的？

## 精选留言 (9)

 写留言



Peter Cheng

2020-04-21

我想请教一个问题。用new Obejct和new Map创建有什么区别。Map有什么优化？



2



一步

2020-04-19

V8 有了隐藏了，所有的查询属性的操作都会走隐藏了吗？这里除了内联缓存，还会走挨个遍历的属性的方法吗？或者什么时候隐藏类会失效，退化为挨个遍历的属性 查找属性？



1



文简

2020-04-18

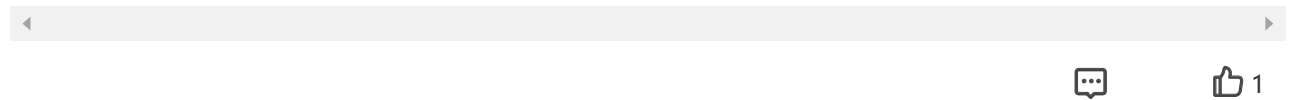
最后的思考题，答案应该是使用『内联缓存』吧。我找到了一篇讲内联缓存的文章，挺好



的 <https://blog.csdn.net/szengtal/article/details/72861133>

展开 ▾

作者回复: 是的, 思考题是关于内联缓存策略的



**yunplane**

2020-04-22

隐藏类中的偏移量都是针对对象的属性是数值的情况吧? 如果对象的属性是字符串或者函数呢?



**潇潇雨歇**

2020-04-19

使用缓存, 执行loadX, 将其隐藏类缓存起来, 再一次调用直接从缓存里查找对象。



**Longerian**

2020-04-19

第3节已经提到了动态添加的数据利用elements 属性和 properties 属性提升属性的访问速度了, 本文又介绍了隐藏类, 那么有了隐藏类, 还需要 elements 属性和 properties 属性的机制吗?



**一步**

2020-04-19

什么情况下两个对象的形状是相同的, 要满足以下两点:  
相同的属性名称  
相等的属性个数

这里还要要求: 相同的属性顺序吧

展开 ▾



**王楚然**

2020-04-18

思考题:

能调用loadX方法的对象, 都需要有x属性。可不可以直接把x属性地址缓存起来, 每次不必

通过隐藏类去查找？



文简

2020-04-18

有个问题，例子中d8打印出来的 x/y 这两字段为什么是在properties里面？按照第三讲，x和y不应该是快属性吗。可能是我理解不到位，烦请老师解惑

展开 ∨

作者回复: 因为是动态添加的，如果你一次把所有属性都写进字面量，就是快属性了



1

