JavaScript如何工作：V8引擎深入探究 + 优化代码的5个技巧

September 8, 2017 - 原文地址：https://blog.sessionstack.com/how-javascript-works-inside-the-v8-engine-5-tips-on-how-to-write-optimized-code-ac089e62b12e

数周之前，我们开始写作一档专栏，旨在深入挖掘JavaScript，希望能真正弄清楚它是怎么工作的。我们认为，如果了解了JavaScript的构建模块，以及它们之间是如何协同工作的，就能写出更好的代码和app。

该专栏的第一篇文章，主要讲了引擎、runtime和调用栈的概要知识。今天这第二篇，我们会深入地研究Google的V8 JS引擎的内部结构。此外，我们还会提供一些快捷的技巧，帮助大家写出更优质的JavaScript代码——这些技巧是我们在SessionStack的开发团队开发产品时所发现的最佳方案。

概述

所谓的**JavaScript引擎**是一个能运行JavaScript代码的程序(program)或解释器(interpreter)。JavaScript引擎可以是一个标准的解释器，也可以是一个将JavaScript编译成某种形式的字节码的即时编译器。

下面是一些正在开发JavaScript引擎的比较流行的工程：

1、V8——开源，Google用C++开发的
2、Rhino——开源，火狐（Mozilla Foundation）完全用Java开发
3、SpiderMonkey——最早的JavaScript引擎，过去在网景浏览器（Netscape Navigator)中使用，今天则在火狐浏览器(Firefox)中使用
4、JavaScriptCore——开源，市场上称作Nitro，由Apple为Safari开发
5、KJS——KDE的引擎，最初由Harri Porten为KDE项目的Konqueror网页浏览器所开发
6、Chakra（JScript9)——IE浏览器
7、Chakra（JavaScript)——Microsoft Edge
8、Nashorn——OpenJDK开源项目的一部分，用的是Oracle Java语言和工具组
9、JerryScript——用于物联网的轻量级引擎

为什么要开发V8引擎?

V8引擎是由Google开发的开源产品，使用C++开发。该引擎在Google Chrome浏览器中使用。和其他的引擎不同，V8还被流行的Node.js runtime使用。

最初，V8被设计用于提升web浏览器内部的JavaScript运行的性能。为了提升速度，V8把JavaScript代码翻译成执行效率更高的机器码（不使用解释器来做这件事）。在执行JavaScript代码时，V8像很多的现代JavaScript引擎——如SpiderMonkey或Rhino（Mozilla）——一样，实现了一个JIT编译器（即时编译器），从而把JavaScript代码编译成机器语言。和其他引擎最主要的差别在于，V8不会生成任何字节码或是中间代码。

V8曾有两个编译器

在5.9版本（今年早些时候发布）的V8出来之前，V8使用两个编译器：
1、full-codegen——一个简单且快的编译器，它能生成简单和运行起来相对慢的机器码
2、Grankshaft——一个相对来说更复杂的（实时）、优化的编译器，生成高度优化的代码

V8引擎在内部还使用相当多的线程：
1、主线程（main线程）做的是我们通常能想到的事情：拿到我们的代码，编译代码，然后执行之
2、同时，还有一个独立的用于编译的线程，这样主线程就能在该独立用于编译的线程优化代码的时候不间断地执行代码
3、一个Pfofiler线程（分析器线程），它能告诉运行环境（runtime）我们在哪些方法上花了大量的时间，以便Grankshaft可以优化这些方法
4、一些处理垃圾回收清理的线程

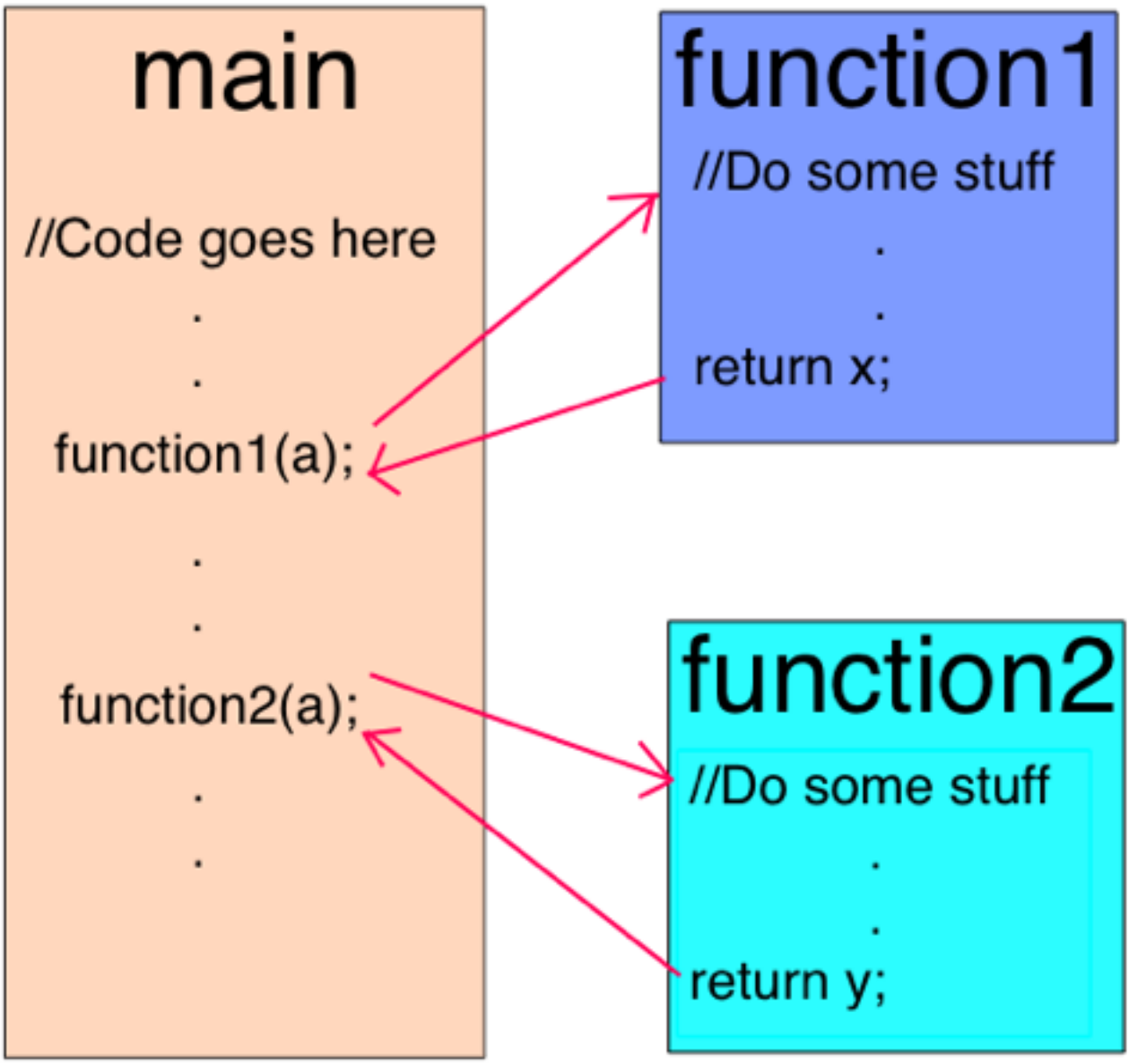
第一次执行JavaScript代码时，V8充分使用full-codegen来将解析过的JavaScript直接翻译成机器码，这个过程不会做任何的中间转化。这种做法使得V8能够非常快速地开始执行机器码。V8不使用中间字节码的表示方式，就没有必要用解释器了。

当我们的代码运行了一段时间后，Profiler线程就会收集到足够的数据，可以判断出哪些方法需要被优化。

接下来，在另一个进程里，Grankshaft优化就开始了。它将JavaScript的抽象语法树翻译成高度静态单赋值的（SSA）表现形式——该表现形式被称为Hydrogen，然后设法优化Hydrogen图。大部分的优化都是在这一层面完成的。

代码嵌入（Inlining）

第一个优化是提前嵌入尽可能多的代码。
代码嵌入（Inlining）是将一个调用点（调用某函数的那行代码）替换成被调用函数的函数体。这个简单的步骤使得接下来的优化更有意义。



隐藏类（Hidden class）

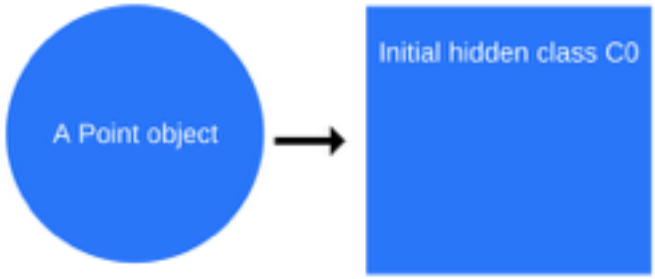
JavaScript是一门基于原型的语言：没有什么类或对象是通过克隆的方式生成的。
JavaScript还是一门动态的编程语言，意味着在一个对象实例化之后，可以轻松地为其增加或移除属性。

大部分的JavaScript解释器使用类似于字典的结构（基于hash函数）存储对象属性值在内存中的位置。这种结构使得相对于非动态编程语言（如Java或C#）而言，在JavaScript中检索一个属性值麻烦很多。Java中，在编译之前，所有对象的属性都由一个固定的对象布局所确定，在运行时不会动态的增加或移除（当然，C#具有动态类型，那是另外一个话题了）。所以，在非动态编程语言中，属性值（或指向属性的指针）在内存中可以被储存在一个连续的buffer里，且两两之间的偏移量是固定的。

由于使用字典在内存中查找对象属性位置非常低效，V8使用了一种不同的方法：隐藏类(hidden classes)。隐藏类和与Java类似的语言中使用的固定对象布局（类）的工作方式非常接近，只是隐藏类是在运行时被创建的。现在，我们就来看看它们到底长什么样：

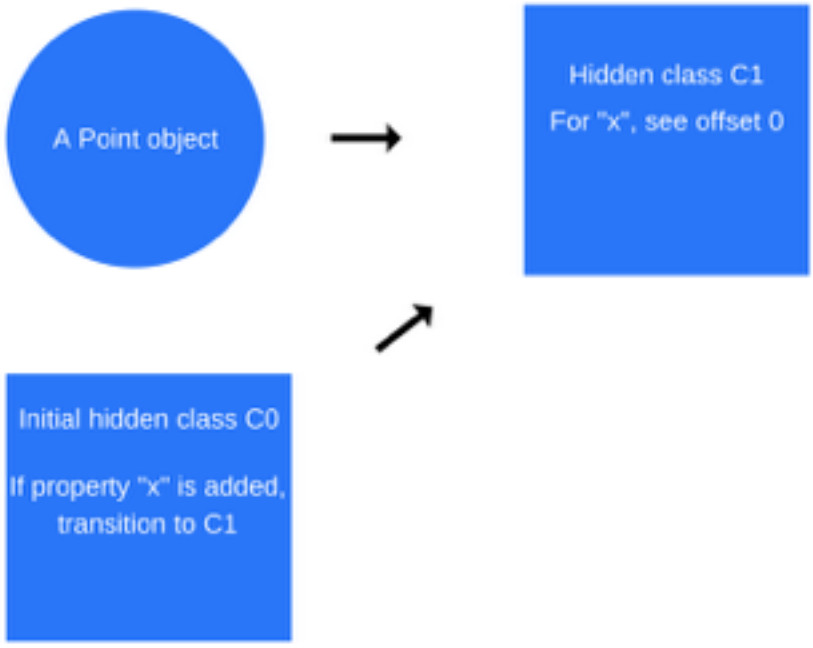
```
function Point(x, y) {
    this.x = x;
    this.y = y;
}

var p1 = new Point(1, 2);
```

一旦“new Point(1,2)”被调用，V8就会创建一个隐藏类，称为“ C_0 ”　。



到目前为止，Point还没有被定义属性，所以“ C_0 ”目前还是空的。

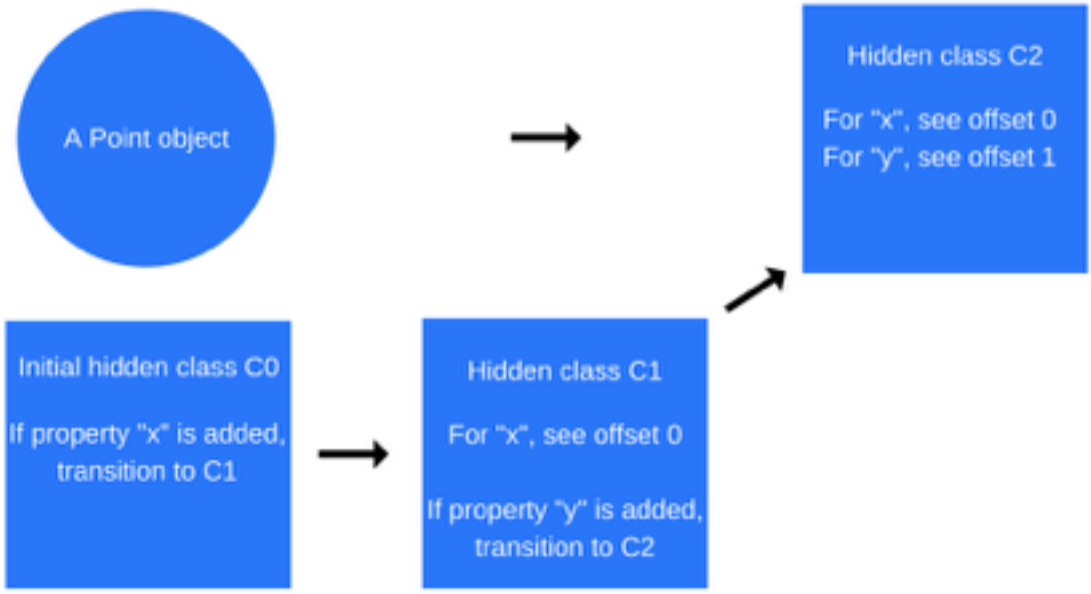
一旦第一个语句“this.x = x”被执行（在“Point” 方法中），V8就会创建基于“ C_0 ”的第二个隐藏类，称为“ C_1 ”。“ C_1 ”描述了在内存中属性x的位置（相对于对象指针的）。在这个例子中，“x”被存储在offset 0，表示在内存中把Point对象视为连续的buffer时，它的第一个offset对应的就是属性 “x”。V8还会用一个 “类转换”对“ C_0 ”做个更新，该 “类转换”描述的是如果一个属性 “x”被添加到一个Point对象上，隐藏类需要从“ C_0 ”变为“ C_1 ”。下面这个Point对象的隐藏类现在就是“ C_1 ”了。



每一次当一个新的属性被添加到某个对象上时，旧的隐藏类就会通过一个转换路径被更新为一个新的隐藏类。“隐藏类转换”非常重要，因为它让相同方式生成的对象们能共享隐藏类。如果两个对象共享一个隐藏类，并且二者都被增加了一个相同的属性，“隐藏类转换”能保证二者能获得相同的新的隐藏类和所有与之关联的优化代码。

当执行 “this.y = y”语句（仍然是Point方法里的；位于“this.x = x”语句之后的那条语句）时，上述过程会被重复一遍。

一个新的名为“ C_2 ”隐藏类被创建，同时一个类转换被添加到“ C_1 ”上——用来描述如果一个属性 “y”被添加到Point对象（其已经包含了属性 “x”）上，那么隐藏类就要变成“ C_2 ”，并且Point对象的隐藏类被更新为“ C_2 ”。



隐藏类转换根据属性被添加到对象上的顺序而发生变化。我们看看下面这一小段代码：

```
function Point(x, y) {
    this.x = x;
    this.y = y;
}

var p1 = new Point(1, 2);
p1.a = 5;
p1.b = 6;

var p2 = new Point(3, 4);
p2.b = 7;
p2.a = 8;
```

你可能会说对p1和p2而言，它们会使用相同的隐藏类和类转换。其实不然~ 对 “p1”来说，先是属性 “a”被添加，然后是属性 “b”。而对 “p2”来说，先是属性 “b”被添加，然后才是属性 “a”。这样， “p1”和 “p2”就在不同的转换路径作用下，有了不同的隐藏类。在这两种情形下，其实最好是用相同的顺序初始化动态属性，这样隐藏类就可以被复用了。

内联缓存（Inline caching）

V8还使用另一种优化动态类型语言的技巧，即所谓的内联缓存。内联缓存的使用，基于我们发现：通常，同一个方法的重复调用是发生在相同类型的对象上的。内联缓存的深度解读可查看这里。

这篇文章我们来说说内联缓存的大致概念。（以防您没有时间阅读上面提到的深度解读文章）

所以内联缓存是怎么工作的呢？V8维护一个对象类型的缓存；这些对象在最近的方法调用中被当做传参，然后V8根据这个缓存信息来推断将来什么样类型的对象会再次被当成传参。如果V8能够准确推断出接下来被传入的对象类型，那么它就能绕开获取对象属性的计算步骤，而只是使用先前查找该对象的隐藏类时所存储的信息。

那么隐藏类和内联缓存的概念是如何关联的呢？当一个特定对象调用一个方法时，V8引擎需要查找这个对象的隐藏类，以便确定获取某个特定属性时的offset。在对于同一个隐藏类两次成功地调用相同的方法后，V8就略去隐藏类的查找，而将这个属性的offset添加到对象自身的指针上。对于未来所有对该方法的调用，V8引擎都假设隐藏类没有发生变化，并使用之前查询中存储的offset值直接跳到特定属性的内存地址里。这个过程极大地提升了执行速度。

内联缓存的使用也是为什么同类型对象共享隐藏类是如此重要的原因。如果我们创建同一个类型的两个对象，而它们隐藏类不同（就如同我们在前面的例子中做的那样），V8就不能使用内联缓存了，因为即使两个对象类型相同，它们对应的隐藏类会给它们的属性分配不同的offset。

这两个对象基本相同，但是“a” 和 “b”属性创建的顺序不同。

编译成机器语言

一旦Hydrogen图被优化，Crankshaft就将这个图降级到一个较低水平的表现形式——称为Lithium。大多数的Lithium实现都是面向特定系统结构的。寄存器分配（Register allocation）发生在这一层面。

最后，Lithium被编译成机器码。然后会发生一些其他的事情，即所谓的OSR：on-stack replacement（堆栈上替换）。当我们开始编译和优化一个明显耗时的方法时，我们很可能之前一直在运行它。V8不会将它之前执行的很慢的代码抛在一边，再重新执行优化后的代码。相反，他会对这些慢代码所拥有的全部上下文（堆栈，寄存器）做一个转换，以便能

够在执行这些慢代码的过程中直接切换到优化后的版本。这是一个非常复杂的任务，要知道，V8已经在其他的优化中将代码嵌入了（inlined the code initially）。当然，V8不是唯一一个能做到这一点的引擎。

我们还有被称为 “去优化”的保障措施，能够做相反的转换，将代码逆转成未优化的代码，防止引擎做的假定不再为真时负面效应的出现。

垃圾回收

说到垃圾回收，V8使用一种传统的分代式标记清除方法（a traditional generational approach of mark-and-sweep），来清除老一代。标记阶段会阻止JavaScript执行过程。为了控制垃圾回收的成本，并使代码执行更稳定，V8使用增量标记：和遍历整个堆(heap)、试图标记所有可能的对象不同，它仅遍历部分堆，然后恢复正常的执行。下一次垃圾回收将从上一次堆遍历停止的地方开始。这就使得每一次正常执行之间的停顿非常短暂。如前文所述，清除操作是由独立的进程来处理的。

点火和涡轮风扇（Ignition and TurboFan）

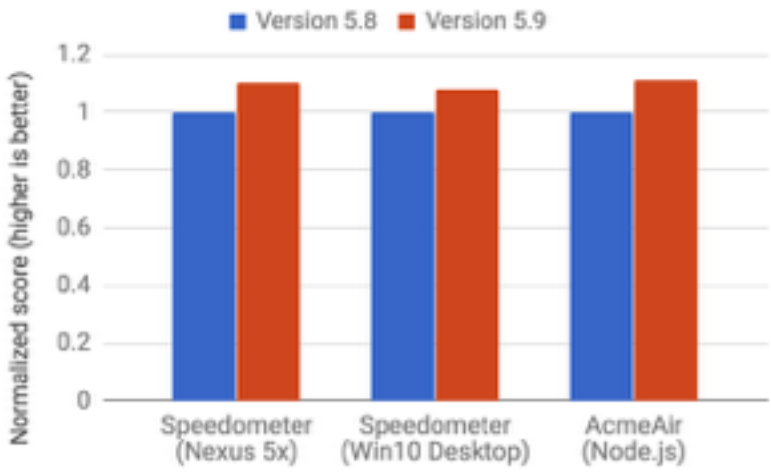
随着2017年早些时候V8 5.9版本的发布，一个新的执行管线（execution pipeline）被引入了。该新型管线在真实世界的JavaScript应用中甚至取得了更大的性能提升和巨大的内存节约。

该新型管线构建于V8解释器Ignition和最新的优化编译器TurboFan之上。

你可以在此查看V8团队有关该主题的博文。

V8 5.9版本问世后，由于V8团队力争和新的JavaScript语言特性以及针对这些新特性所需要的优化保持一致，full-codegen和Crankshaft（这两项技术从2010年开始为V8服务）不再被V8用来运行JavaScript。

这意味着整个V8将拥有更简单和更易维护的架构。



Web和Node.js基准上的改进

这些改进只是一个开始。新的Ignition和TurboFan管线为未来的优化铺平了道路，未来JavaScript的性能会有更加巨大的提升，并能让V8在Chrome和Node.js中节约资源。

最后，这里提供一些小技巧，帮助大家写出更优化的、更优质的JavaScript。从上文中您一定可以轻松地总结出一些技巧，不过为了方便，仍然为您提供一份总结。

如何写出优化的JavaScript

1、对象属性的顺序：永远用相同的顺序为您的对象属性实例化，这样隐藏类和随后的优化代码才能共享。

2、动态属性：在对象实例化后为其新增属性会导致隐藏类变化，从而会减慢为旧隐藏类所优化的方法的执行。所以，尽量在构造函数中分配对象的所有属性。

3、方法：重复执行相同方法的代码会比不同的方法只执行一次的代码运行得更快（由于内联缓存）。

4、数组：避免使用keys不是递增数字的稀疏数组（sparse arrays）。并不为每个元素分配内存的稀疏数组实质上是一个hash表。这种数组中的元素比通常数组的元素会花销更大才能获取到。此外，避免使用预申请的大型数组。最好随着需要慢慢增加数组的大小。最后，不要删除数组中的元素，因这会使得keys变得稀疏。

5、标记值（Tagged values）：V8用32个比特来表示对象和数字。它使用1个比特来区分是一个对象（flag = 1）还是一个整型（flag = 0）（被称为SMI或SMall Integer，小整型，因其只有31比特来表示值）。然后，如果一个数值大于31比特，V8就会给这个数字进行装箱操作（boxing），将其变成double型，并创建一个新的对象将这个double型数字放入其中。所以，为了避免代价很高的boxing操作，尽量使用31比特的有符号数。

参考资源：

https://docs.google.com/document/u/1/d/1hOaE7vbwdLLXWj3C8hTnnkpE0qSa2P--dtDvwXXEeD0/pub

https://github.com/thlorenz/v8-perf

http://code.google.com/p/v8/wiki/UsingGit

http://mrale.ph/v8/resources.html

https://www.youtube.com/watch?v=UJPdhx5zTaw

https://www.youtube.com/watch?v=hWhMKalEicY