V8 之旅：full compiler

在过去的五年中，JavaScript的性能有了极大的提升，这主要归功于JavaScript虚拟机的执行机制由解释演变为了JIT。现在，JavaScript成为了HTML5的中坚力量，推动着新一波Web技术的发展。JavaScript引擎中，V8是最早使用原生代码的引擎之一。V8现已成为了Google Chrome、Android浏览器、WebOS及Node.js这样的其他项目中不可分割的重要组件。

本文来自Jay Conrod的A tour of V8: full compiler，其中的术语、代码请以原文为准。

一年多前，我（指的是原作者）进入了我们公司的一个负责V8在我们ARM产品上优化的团队。从那时算起，由于软硬件性能的提升，我已亲眼见到SunSpider性能翻倍，V8性能测试提升近50%。

V8是一个非常有趣的项目，然而它的文档却非常分散。在接下来的几篇文章中，我将在较高的层面上对其做一个概述，希望对其他同样对VM或编译器内部原理感兴趣的朋友们能有所帮助。

全局架构

V8将所有JavaScript代码编译为原生代码执行，其中没有任何的解释器以及字节码参与。编译以函数为单位，一次编译一个（这与FireFox VM原有的TraceMonkey引擎相反，TraceMonkey为追踪式编译，并不以函数为单位）。通常，函数在初次调用之前是不会被编译的，因此如果你引用了一个大型的脚本库，VM并不会花大量的时间去编译那些根本没用到的部分。

V8实际上有两个不同的JavaScript编译器。我个人喜欢将其看作一个简单编译器及一个辅助编译器（*译注，这里看起来没有一个正经的，但实际上两个词汇描述的方面不同。前者指的是机制简单的编译器，后者指的是使用频度低的编译器。*）。Full Compiler（*对应简单编译器*）*是一个不含优化的编译器，其工作就是尽快生成原生代码，以保持页面始终快速运转。Crankshaft（对应辅助编译器*）则是一个带有优化能力的编译器。V8会将任何初次遇到的代码使用FC编译，之后再使用内置的性能分析器挑选频度高的函数，使用Crankshaft优化。由于V8基本上是单线程的（截至3.14版），任何一个编译器运行时，都会打断脚本的执行。在V8未来的版本中，Crankshaft（或者至少其中一部分）将会在一个单独的线程中运行，与JavaScript的执行并发，以便进行更多昂贵的优化。

为何没有字节码?

大多数VM都有一个字节码解释器，但V8却没有。你可能好奇为何原本应当先编译为字节码再执行的过程，被FC替换掉了。原因是，编译为原生代码并不会比编译为字节码耗去太多。考虑如下两个过程：

在上述两个过程中，我们都需要解析源码以及生成抽象语法树（AST），我们都需要进行作用域分析，以便得出每个符号所代表的是局部变量，上下文变量（闭包相关）或全局属性。唯独

转换的过程是不同的。你可以在这一步做一些非常细致的工作，但你也同时希望编译器越快越好，甚至很想来个“直译”：语法树的每个节点都转化为一串相应的字节码或原生代码指令（*译注，汇编指令*）。

现在思考一下你会如何去做一个字节码解释器。一个朴素的实现可能就是一个循环，其中会不断获取字节码，然后进入一个大的`switch`语句，逐一执行其事先准备好的指令。有一些途径对这个过程进行改进，但最终还是会落到相近的结构上。

如果我们此时不是去生成字节码、使用解释器的那个循环，而是直接触发相应的原生代码呢？无需如果，V8的FC就是这样做的。这样做便不再需要解释器，并且大大简化了未优化代码与优化代码之间的切换。

一般来说，字节码发挥用武之地的最佳时机，是编译器有充分的准备时间的时候。但这并不是浏览器中所能允许的，因此FC对于V8来说更加应景。

内联缓存：加速未优化代码

如果你看过ECMAScript标准，你会发现其中有很多操作异常复杂。以+操作符来说，如果操作数都为数字，则它演绎为加法；如果其中有一个操作数是字符串，则它演绎为字符串拼接；如果操作数不是数字也不是字符串，其将经过某些复杂的（可能是用户定义的）过程，转化为原语（*译注，原语指的是JavaScript中的数字、字符串、布尔、`undefined`以及`null`*），最终再演绎为数字加法或字符串拼接。仅仅是查看脚本源码，我们无从得知哪种操作最终应当执行。属性的读取（比如：`o.x`）是另一个潜在复杂操作的例子。只通过源码，你将无从得知你要的是读取一个对象自己的属性（对象本身所具有的属性），还是原型对象的属性（来自于原型链上原型的属性），还是一个`getter`方法，亦或是浏览器的某些自定义回调。这个属性还可能根本不存在。如果你要在FC编译的代码中处理所有这些情况，即使一个简单的操作也会引发上百条指令。

内联缓存（Inline caches， ICs）提供了一个优雅的方案来解决这个问题。内联缓存大致就是一个包含多种可能的实现（通常运行时生成）来处理某个操作的函数（*译注：拗口，我的理解是，这个函数提供了多个处理问题的方案，这些方案的性能由优至次，一个不行就退化到另一个，直至最终最低效率的方法*）。我之前曾写过函数的多态内联缓存的文章。V8使用IC处理了大量的操作：FC使用IC来实现读取、存储、函数调用、二元运算符、一元运算符、比较运算符以及`ToBoolean`隐操作符。

IC的实现称为Stub。Stub在使用层面上像函数：调用、返回。但它不必初始化一个调用栈来完成调用约定。Stub常常在运行时动态生成，但在通常情况下都可被缓存，并被多个IC重用。Stub一般会含有已优化的代码，来处理某个IC之前所碰到的特定类型的操作。一旦Stub碰到了优化代码无法解决的操作，它会调用C++运行时代码来进行处理。运行时代码处理了这个操作之后，会生成一个新的Stub，包含解决这个操作的方案（当然也包括之前的其他方案）。对原有Stub的调用随即变为了新Stub的调用，脚本的执行也将继续进行，变得和Stub正常的调用流程一样。

我们来看一段简单的例子，读取属性：

```
function f(o) {
  return o.x;
}
```

当FC初次生成代码时，它会使用一个IC来演绎这个读取。IC以uninitialized状态（初态）初始，调用一个不包含任何优化代码的简易的Stub。下面是FC生成的调用stub的代码：

```
;; FC调用
ldr   r0, [fp, #+8]     ; 从栈中读取参数”o“
ldr   r2, [pc, #+84]    ; 从固定的位置读取”x“
ldr   ip, [pc, #+84]    ; 从固定位置载入uninitialized态的stub
blx   ip                ; 调用stub
...
dd    0xabcdef01        ; 上面拿到的stub地址
                        ; 当stub出现处理不了的操作时，这里的stub会被换成新的stub
```

（如果你不熟悉ARM汇编的话，抱歉。希望注释能让代码的意图清晰）
这是处于uninitialized态的stub：

```
;; uninitialized stub
ldr   ip,  [pc, #8]   ; 读取C++运行时的函数来处理
bx    ip              ; 尾调；译注：尾递归优化技术
...
```

当stub第一次被调用时，stub注定无法处理它所面对的操作，运行时代码会替stub来解决。在V8中，最常见的存储属性的方法就是将其放在对象中一个固定偏移量的地方，我们以此为例。每个对象都有一个指向Map的指针，也即一个描述对象布局的一个不变结构。负责读取对象自身属性的stub会将对象的布局图与已知的Map（也就是运行时所生成的Map）相比较，来快速确定对象是否在相应的位置存放着该属性。这个Map的检查使我们能够避开一次麻烦的Hash表查询。

```
;; monomorphic态的对象自身属性读取stub
tst   r0,   #1          ; 检验目标是否是一个对象；译注：见代码末详细译注
beq   miss              ; 不是就说明处理不了
ldr   r1,   [r0, #-1]   ; 读取对象的Map
ldr   ip,   [pc, #+24]  ; 读取已知的Map
cmp   r1,   ip          ; 它们相同否?
bne   miss              ; 不同说明处理不了
ldr   r0,   [r0, #+11]  ; 读取属性
bx    lr                ; 返回
miss:
ldr   ip,   [pc, #+8]   ; 调用C++运行时来解决
bx    ip                ; 尾调
...
```

译注：V8中对32bits长的值做了进一步分类，其中最低位作为区分，如果为0则表示该值为31bits长的整数；如果为1则表示该值为30bits长的指针。由于V8中的对象以4Bytes为单位对齐，指针的最低2位恰好空闲。

只要该表达式只负责读取对象自身的属性，则读取可以无附加地快速完成。由于IC只处理了一种情况，它处于monomorphic态（单态）。如果在后续的运行中，这个IC又遇到了无法处理的情况，则更加常见的megamorphic态（复态）stub会被生成。

待续...

如上所述，FC圆满地完成了它快速生成优质代码的任务。由于IC易于扩展的特点，FC生成的代码也非常通用，这使得FC非常简单；而IC则使代码非常灵活，能够处理任何情况。

在接下来的文章中，我们将看到V8内部如何表达JavaScript对象，来做到在大多数场景下以O(1)的时间访问这些程序员未做任何结构定义工作（类似于类定义）的对象。