V8 之旅：优化编译器 Crankshaft

在之前的两篇文章中，我们讨论了V8的Full Compiler和对象的内部表示。在几年前，FC生成的原生代码相对于JavaScript来说已经不错了，但人们对性能的要求与日俱增，其速度标杆也越来越高，因此衍生出了Crankshaft。

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

Crankshaft是V8的优化编译器。回忆一下，V8有两个编译器，另一个编译器FC负责尽快生成未优化的代码。对于只执行几次的代码，FC生成的代码还是比较理想的。当FC产生的代码运行过一段时间之后，V8会挑选出“热门”的函数，重新用Crankshaft编译。这大大提升了性能。

热身完毕

如果只看脚本，很难说哪个函数最应当得到优化。V8使用一个运行时性能分析器在脚本运行的时候识别热门的函数。

Crankshaft刚开始部署时，V8选择在另一线程跑栈采样分析器（stack sampling profiler）。几乎每隔一段时间（桌面版本为1ms，移动版本为5ms）那个线程就被唤醒一次，然后向主线程发送`SIGPROF`信号。主线程的信号处理代码会重置栈顶的高度（一般是一个标志栈结束的地址）。每当一个函数被调用以及每轮循环的时候，经过JIT的代码会检查是否达到栈顶，如果栈指针超过了栈顶，就会调用运行时来报错。这给了性能分析器一个打断脚本执行的时机。V8运行时会在检查出栈溢出是因为信号`SIGPROF`时调用性能分析器，于是性能分析器就可以在这时看到栈顶的几帧，然后标注这些函数进而优化。

这种性能分析器有几个短板。由于采样是几近随机的，性能因素并不主导采样。尽管分析器在统计上倾向于挑选出最热门的函数，但其可能在页面每次重载时得出不同顺序或不同次数的结果。想象一下当时的情景，性能测试的时候得出的是差异很大的结果，V8测试集当中的某些测试甚至能在多次运行时差距达50%。同时这种方案会因其打断代码执行的机制，在整体上对不含循环的大函数有失偏颇。

V8如今用基于计数的性能分析器（counter-based profiler）。每个经过FC的函数都包含一个计数器，当函数返回或完成一轮循环的时候，就会减少计数的值。而减多少则依据函数或循环的大小，因此这对于大函数和循环来说更加公平。分析器在计数减到0的时候调用，然后和栈采样分析器类似（实际上更加出色），但更侧重性能地选出热门函数。另外这样对于已优化的代码来说没有任何影响，因为只有未优化的代码才会有计数器。

一旦一个函数被分析器标记为需要优化，指向其代码的指针就会被改写指向为一个V8内置的函数——LazyRecompile，来调用编译器。这样函数就会在下次调用时得到优化。

剖析Crankshaft

Crankshaft经过以下几个阶段生成代码：

- 语法分析：这一阶段负责将源代码翻译为AST。Crankshaft与FC共享同一个语法分析器，但出于空间占用考虑，V8并不保留任何编译器所得到的AST（以及其他中间产物）。而且AST也不常用，生成也很容易。
- 作用域分析：在这一阶段，V8将确定变量是如何被使用的，将其与各自的定义链接。局部变量、闭包变量、全局变量的对待方式会各不相同。这个阶段也与FC共享。某些代码的写法（比如用`eval`动态引入变量）会使一个函数失去优化或内联的资格。
- 图生成：Crankshaft使用AST、作用域信息以及FC代码反馈而来的类型信息，来构建Hydrogen控制流程图。内联也发生在这一阶段。Hydrogen是Crankshaft中一个高层次、架构无关的中间代码。
- 优化：绝大多数优化都基于Hydrogen控制流程图发生在这一阶段。这是Crankshaft唯一能够与JS线程并行的时候。虽然本文撰写时还不是并行的。
- 低级化：优化结束之后，Hydrogen外会生成一个Lithium图。Lithium图是一个低级且架构相关的中间代码。寄存器的分配就是在这一阶段施行的。
- 代码生成：在这个最后阶段中，Crankshaft按照Lithium图中的每个指令发送原生指令。元数据，比如重定位信息以及去优化数据，也是在这一阶段生成。完成之后，经过JIT的代码会放在一个Code对象中，然后继续脚本的执行。

整个结构对于优化编译器来说非常典型，然而某些方面可能会让你惊讶。首先，Crankshaft并没有一个真正的低级表达形式。Hydrogen和Lithium的指令基本上和JS中的操作相对应。在某些情况下，十来个原生指令才对应一个Lithium指令。这可能会导致生成出来的代码中有一定的冗余。第二，Crankshaft一种指令调度都没有。对于x86处理器来说不算大问题，因其最终是乱序执行；但对于一些相对简单的RISC指令集架构，比如ARM，会造成一些麻烦。

这些缺点大多是因为Crankshaft在性能影响上的重要地位。JS代码在Crankshaft执行时是中断的，这就意味着Crankshaft必须尽快完成任务，而任何附加的阶段和优化都可能得不偿失。V8的开发者们正在为Crankshaft的并行上努力，但这一功能目前还没有打开。V8的堆并不支持多线程访问，而如果不访问堆，则只有优化阶段能够并行。然而使堆变得线程安全是一项复杂的任务，可能会引入额外的同步负担。

Hydrogen与Lithium

V8的高级中间代码叫做Hydrogen，而低级中间代码叫Lithium。如果你曾经用过LLVM，Hydrogen的结构看起来会有些熟悉。函数被表达为一个由一组区块构成的流程图，其中的每个区块都包含一系列<a href="http://en additional resources.wikipedia.org/wiki/Static_single_assignment_form”>静态单赋值形式（SSA）的指令。每条指令则包含一组操作符和一组操作符调用，因此你可以将其想象为一个叠在流程图之上的数据流图。每个Hydrogen指令表示一个较为高级的操作，比如算术运算、属性的存/取、函数调用或者类型检查。

大多数优化过程发生在Hydrogen身上或构造Hydrogen的时候。这是Crankshaft所执行的优化：

- 动态类型反馈：在图生成的同时，大多数操作会特化为对一个类型的操作。这些类型在FC代码的内联缓存中得到。比如，如果IC实现的是一个读取操作，而这个读取

只作用于一种对象的某个属性，特化的Hydrogen代码会将被优化为只处理这一种对象的代码。为此，必要时会加入类型检查。

- 内联：发生在图的生成阶段。内联的启发规则非常简单：基本上如果一个函数在调用时已知且内联它是安全的，那么就内联它。大函数（源码中大于600个包括空格在内的字符，或超过196个AST节点）则不会内联。最多只有196个语法节点可在一个函数中内联。
- 形式推断：Hydrogen支持三种寄存器中的值：标记值、整型值和双精度浮点。所谓标记值，就是指这个值是封箱的。字符串和对象总是标记值，但数字则不一定。原生整型和双精度浮点更有效率，但所有的值都必须在存到内存中或传递给另一个函数前标记。这一环决定了每个值应有的表达形式。
- 静态类型推断：这一步Crankshaft尝试确定函数中各种值的类型。由于一次只能针对一个函数而且大多数操作（比如函数调用、属性读取）产生的类型无法推断，这步效率很低。不过有些类型检查会因为该值有精确的类型信息而省去。
- UInt32分析：V8使用31bits来表示小整数。其最低位保留给垃圾回收器用以判定它是指针还是数字（0表示数字，1表示指针）。Crankshaft可以使用全部32-bit来保存不经过内存的局部变量和临时值，而当超出这一区间时，则需要特殊处理。语义上，JavaScript将所有数字都视为64位双精度浮点，因此允许用整数来表达原本是错误的。但这一步将某些操作归纳为无符号的，于是微小的溢出并不会在这些特定情况下造成麻烦。这对于诸如密码学、压缩和图形处理相关的程序来说非常有用。
- 标准化：这步是用来进行简化的。它去掉了不必要的操作，并进行一些简化。
- 值编号（GVN）：这是去除冗余的标准步骤。依次处理每个指令，当一个指令处理之后，会生成一个基于其操作、输入以及任何相关数据的哈希值，插入到一个哈希表中。后续如果遇到同样哈希值的指令，则GVN会删除后续的那个。但每当遇到对该指令存在副作用的指令，则从哈希表中清除原先存入的这个指令。比如，对于两个完全一样的读取操作来说，如果中间还有一个存储操作，则这两个读取操作并不能合并。

  移出循环无关代码（LICM）：这个和GVN同时执行。循环中并不依赖循环中其他代码的指令，将被提到循环之前。对循环无关值的类型检查也会被提到循环之前，但这可能会导致某些场景下并不会发生的类型检查也被执行，因此编译器会在这时趋于保守。
- 范围分析：这步会确定每个整数操作的上限和下限。这样就有可能去掉一些溢出检查。在实践中，这个并不太精确。
- 去除冗余的数组范围检查：去掉某些已在数组元素访问中执行过的多余数组范围检查。
- 后推数组下标计算：这一步会反转LICM对一些非常简单的表达式（如数组下标增加或减去一个常数）的效果。所有V8支持的架构都有指令来完成寻址时对下标的简单加减，因此提前这些代码通常也没什么大的好处。
- 去除无效代码：这是一种清理工作。它会移除掉Hydrogen指令中无效或者没有副作用的部分。这些指令往往是其他优化所产生的副产品，而程序员自己所写的无效代码则不包含在内：V8可能会在函数运行的过程当中，在优化代码和未优化代码之间切换，而如果优化后的代码缺失了某个未优化代码所需要的值，则可能会造成崩溃。（*译注：也就是说，在Crankshaft看来可能无效的代码，很可能只是整个脚本的一部分，而整个脚本中的其它未优化部分，实际是需要那部分看似无效的代码的。*）

Lithium是V8的低级、机器相关的中间代码。实际上它并不是特别的低级：每个Hydrogen指令至多被低级化为一个Lithium指令。有些Hydrogen指令，例如HConstant、HParameter原封不动变成了Lithium指令。其他一些Hydrogen指令会被低级化为某些由操作数衔接起来的指令（而在Hydrogen中它们本是直接相连的）。这里的操作数可能是常数、寄存器，或者栈槽。寄存器分配器将决定每个操作数的类型和其存放位置。大多数指令只支持寄存器操作数，因此寄存器分配器需要在这些操作数中间增加存取的指令。

原生代码将从Lithium指令得出。简单的Lithium指令可能只对应一条原生指令，而某些复杂的Lithium指令则会对应10-20条原生指令（ARM如此，其他架构可能有差异）。

即使是在只处理常规使用的优化代码当中，你也会因JavaScript脚本最终产生的复杂代码而咋舌。举例来说，以下是为一个JS对象增加一个属性的代码：

```
        ;; HCheckNonSmi: 检查低位，确定目标是整数
        ;; 整数的最低位总是0
         40  tst r0, #1
         44  beq +188 -> 232

        ;; HCheckMaps: 将目标的Map与已知Map对比
         48  ldr r9, [r0, #-1]
         52  movw ip, #41857             ;; object:
         56  movt ip, #17120
         60  cmp r9, ip
         64  bne +172 -> 236

        ;; HCheckPrototypeMaps: 检查目标的原型链，
        ;; 以防有同名的只读属性
         68  movw r2, #38241             ;; object:  Cell for
         72  movt r2, #15696
         76  ldr r2, [r2, #+3]
         80  ldr r1, [r2, #-1]
         84  movw ip, #41937             ;; object:
         88  movt ip, #17120
         92  cmp r1, ip
         96  bne +144 -> 240
        100  movw r2, #46869             ;; object:
        104  movt r2, #14674
        108  ldr r1, [r2, #-1]
        112  movw ip, #36057             ;; object:
        116  movt ip, #17120
        120  cmp r1, ip
        124  bne +116 -> 240

        ;; HConstant: 这才是我们要存储的值
        128  mov r1, #198

        ;; HStoreNamedField: 以下是存储操作
        ;; 首先，读取Transition的Map
        132  movw r9, #41977             ;; object:
        136  movt r9, #17120

        ;; 接着，将其存入该对象的首字
        140  str r9, [r0, #-1]

        ;; 可能还需要将Map的变动通知垃圾回收器
        ;; 这里是一个写操作的内存屏障，
        ;; 递增标记需要这样的保证
        144  sub r2, r0, #1
        148  bfc r9, #0, #20
        152  ldr r9, [r9, #+12]
```

```
156  tst r9, #4
160  beq +36 -> 196
164  mov r9, r0
168  bfc r9, #0, #20
172  ldr r9, [r9, #+12]
176  tst r9, #8
180  beq +16 -> 196
184  movw ip, #37056
188  movt ip, #10611
192  blx ip

;; 现在我们终于可以存储那个属性了
;; 因为这次存储的是整数，无需通知垃圾回收器
;; 但通常这里还需要另一个内存屏障
196  str r1, [r0, #+11]
```

栈上替换

我们讲性能分析器时说过，经过分析器标记的函数，编译器会在下一次调用时对其进行优化编译，而当编译工作结束后，运行时即会载入新的优化代码来执行。对于大多数函数来说，这种机制已经足够好了，但对于包含热门循环，且只运行一次的函数来说，这个机制就有瑕疵了。

```
function calledOnce() {
  for (var i = 0; i < 100000; i++)
    // ... 这里是需要优化的部分
}
```

这类函数仍然需要优化，因此一种略复杂的机制应运而生。如果性能分析器被触发时，一个函数已被标记为需要优化但还没有再次被调用，则分析器会尝试栈上替换。分析器会直接调用Crankshaft，立即生成出优化代码；当Crankshaft执行完毕时，V8会依据原先包含该函数的栈帧中的其他代码，构造一个新的栈帧来存放优化后的代码。这时将旧的栈帧弹出，压入新的栈帧，恢复脚本的运行。

这一过程需要两个编译器的支持。FC需要产生包含该栈帧中其他值的位置信息，而Crankshaft需要生成这些值即将存放的新位置的信息。同时，Crankshaft还需要生成额外的“接入层”，来将这些值从栈帧读取到正确的寄存器当中。

去优化

上面提到过，Crankshaft生成的代码只是特化针对于FC所遇到的类型，因此Crankshaft无法处理所有可能的值。我们需要一种机制来在遇到未知类型和算术运算溢出时优雅降级，换用FC的代码。这种机制就叫做去优化。通常去优化就是进行栈上替换的逆操作。然而这里面有个小问题：Crankshaft支持内联，因此类型问题有可能会在内联代码之内发生。这时去优化过程将不得不对多个栈帧进行操作。

为了使去优化器的工作更加容易，Crankshaft会生成去优化的输入数据，每个去优化可能发生的地方，都会有相应的命令关联。去优化器将通过这些命令，来将寄存器、栈槽等从优化代码中的值，转换为未优化代码中的值。每条命令都包含有与栈相关的操作，比如“从寄存器r6中获取值，将其封箱为数字，然后压入下一栈槽”。去优化器的任务就是寻找正确的命令，将其执行，然后弹出优化的栈帧，压入相应未优化的栈帧。

总结

Crankshaft是V8的秘密武器。通过运行时的性能分析器，V8能够检测出最需要优化的函数。由于V8的去优化随时可能需要将代码降为未优化代码，Crankshaft可以在一定程度上具有推断能力，只优化特定情况下的代码。

在将来，Crankshaft很可能会并行。由于可以腾出更多的时间优化更多的代码，这能够让V8的性能更高。