ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences

智能软件研究中心 Intelligent Software Research Center

V8中HelloWorld的解释执行过程-part1

智能软件研究中心 邱吉

qiuji@iscas.ac.cn

2021/08/15

内容

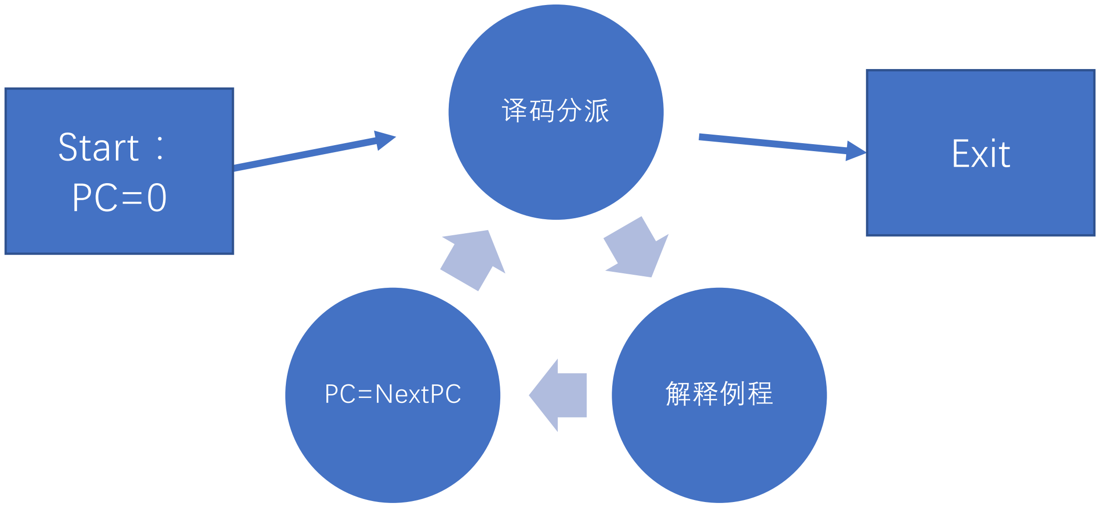
- 背景知识：
  - 需要了解的前序知识
  - 解释器基础概念
  - V8的整体架构，解释器Ignition和其他部分的关系
- hello.js：print(“HelloWorld!”)的字节码和含义
- hello.js的解释执行过程
- 调试的代码和log：https://github.com/qjivy/v8/tree/v8ignition-learn

前序知识

- V8的Simulation build和running的含义：使用V8的内置riscv64指令集模拟器，在x64上运行riscv64的解释器和turbofan- https://www.bilibili.com/video/BV1HV41167Af by 陶立强
- V8 Ignition 解释器的工作过程：https://www.bilibili.com/video/BV16b4y1f7Wf by 刘铮
- V8的Builtin和code stub assembler： https://v8.dev/docs/csa-builtins

背景知识

- 解释器是什么？
  - 在虚拟机中，解释器是将源ISA的程序，通过逐条指令地进行“译码分派”，“解释例程执行”，“程序计数器递增”的执行流程，来实现源ISA的仿真的程序



  - 解释器广泛地存在于各种形式的虚拟机中，比如指令集仿真器、二进制翻译器、高级语言虚拟机等
  - 相比于JIT，解释器无启动开销，通常用于初始阶段的运行并同时搜集热点

背景知识

- 解释器的实现方法
  - 简单的译码分派：译码分派通过switch-case来实现，在case语句中调用解释例程。switch-case和调用会产生大量直接和间接分支指令，导致仿真器运行的平台宿主机需要大量CPU cycle来预测和处理，性能会较差
  - 线索解释：首先建立操作码和解释例程地址的映射表，然后，在解释例程的末尾处直接进行NextPC的译码，根据opcode在映射表中查找解释例程地址，直接跳转到对应的解释例程处进行下一条指令的仿真，从而省去了switch-case结构。仿佛把解释例程用线串在了一起，中间不再回到switch-case的大循环。

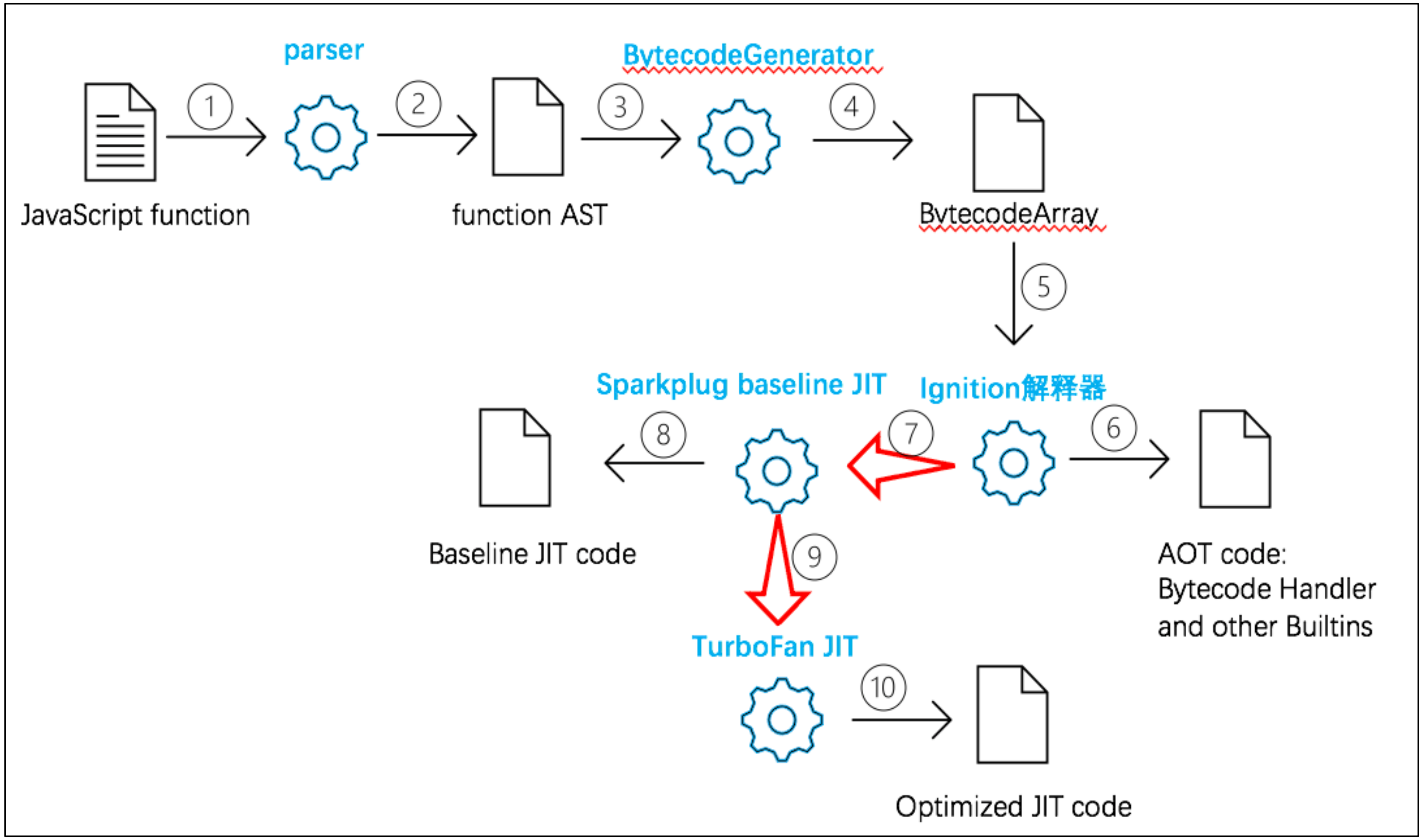
```
while (true) {
   switch (opcode) {
Switch statementwhile (true) {
         ...
         break;
      case SUB:
         ...
         break;
      ...
   }
}
```

```
typedef void *Inst;
Inst program[] = { &&ADD, &&SUB };
Inst *ip = program;
goto *ip++;

ADD:
    ...
    goto *ip++;

SUB:
    ...
    goto *ip++;
```

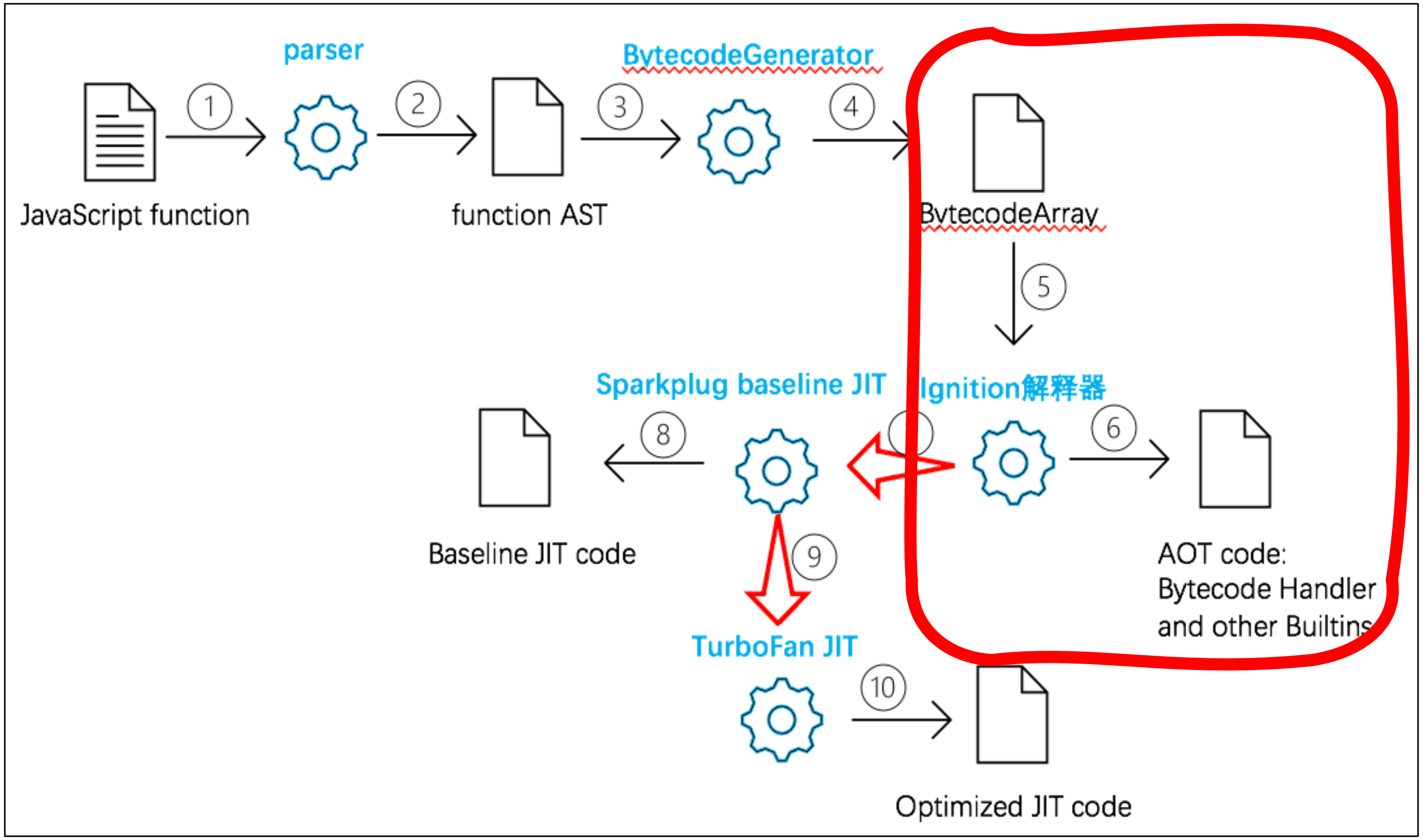
V8的解释器实现是线索化的

https://www.slideshare.net/nwind/virtual-machine-and-javascript-engine

V8 compilation pipeline overview



V8 compilation pipeline overview



今天的内容：

1. hello.js的bytecodeArray里有什么？
2. Ignition是如何执行的？

hello.js的字节码

./d8 --print-bytecode hello.js

```
print( " hello" )
```

```
Bytecode length: 13
Parameter count 1
Register count 3
Frame size 24
OSR nesting level: 0
Bytecode Age: 0
        0xdf23ca206e @    0 : 21 00 00         LdaGlobal [0], [0]
        0xdf23ca2071 @    3 : c2               Star1
        0xdf23ca2072 @    4 : 13 01            LdaConstant [1]
        0xdf23ca2074 @    6 : c1               Star2
        0xdf23ca2075 @    7 : 61 f9 f8 02      CallUndefinedReceiver1 r1, r2, [2]
        0xdf23ca2079 @   11 : c3               Star0
        0xdf23ca207a @   12 : a8               Return
Constant pool (size = 2)
0xdf23ca2019: [FixedArray] in OldSpace
 - map: 0x001bf11012c1 <Map>
 - length: 2
           0: 0x00df23c813a9 <String[5]: #print>
           1: 0x00df23ca1f71 <String[5]: #hello>
Handler Table (size = 0)
Source Position Table (size = 0)
```

hello.js的字节码

LdaGlobal [0], [0]	LdaGlobal <name_index> <slot> Load the global with name in constant pool entry <name_index> into the accumulator using FeedBackVector slot <slot>.
Star1	Store the accumulator into r1.
LdaConstant [1]	LdaConstant <idx> Load constant literal at |idx| in the constant pool into the accumulator.
Star2	Store the accumulator into r2.
CallUndefinedReceiver1 r1, r2, [2]	Call <callable> <receiver> <arg_count> <feedback_slot_id> Call a JSfunction or Callable in |callable| with the |receiver| and |arg_count| arguments in subsequent registers. Collect type feedback into |feedback_slot_id|
Star0	Store the accumulator into r0.
Return	Return the value in the accumulator.

- 字节码的含义和格式，可以参考src/interpreter/interpreter-generator.cc 文件
- 蓝色的部分是feedback vector slot的索引号，用于字节码执行时类型信息的记录。如果只讨论解释器的执行流程，可以忽略掉它们

hello.js的解释执行流程

- 分析内容：
  - 解释器是如何进入的？（今天的内容）
  - 如何译码分派的？（后续课程）
  - 解释例程在哪里？如何执行？（后续课程）
- 分析工具：
  - d8 的 --print-bytecode, --trace-sim ， --print-all-code， --trace-ignition(with gn arg v8_enable_trace_ignition=true) 命令行选项
  - d8 的code stub assembler的Print()函数，能够在AOT的code stub中插入字符串常量打印，并在执行时打印出执行流
- 分析方法：
  - 加打印，看代码，分析log和trace

概览./d8 --trace-sim hello.js 2>&1 |tee logtracesim.txt

中国科学院软件研究所 Institute of Software Chinese Academy of Sciences 智能软件研究中心 Intelligent Software Research Center

- 打印的log记录了simulator模式下，V8执行所有riscv64指令的过程，也提供了内置Builtin和host function的调用和返回的信息，包括指令PC，累计执行数量，指令写入的寄存器值，调用的函数名，参数寄存器和返回寄存器值
- 总共执行了795条指令：

```
0x560687d65a1c      06013083      ld       ra, 96(sp)        ffffffffffffffe    (757)    int64:-2 uint64:18446744073709551614 <-- [addr: 7f4f41c92fb8]
0x560687d65a20      06810113      addi     sp, sp, 104       00007f4f41c92fc0   (758)    int64:139978382847936 uint64:139978382847936
0x560687d65a24      00008067      ret                        0000000000000000   (759)    int64:0 uint64:0
```

- "CallImpl"表明的是V8的执行流第一次进入模拟器，进入到了JSEntry这个Builtin中

```
CallImpl: reg_arg_count = 6 entry-pc (JSEntry) = 0x561c340c0840 a0 (Isolate) = 0x561c35f34d20 a1 (orig_func/new_target) = 0xe0e5501599 a2 (func/target) = 0xa7918a1c19 a3 (r
eiver) = 0xa7918838a9 a4 (argc) = 0x0 a5 (argv) = 0x0
  0x561c340c0840      f9810113      addi      sp, sp, -104        00007fdf3e6e6f58    (1)    int64:140596801859416 uint64:140596801859416
  0x561c340c0844      06113023      sd        ra, 96(sp)                              (2)    int64:-2 uint64:18446744073709551614 --> [addr: 7fdf3e6e6fb8]
```

- grep搜索所有的“Call to” 和 “Return to” 的行，就可以得到执行流如何在各个Builtin中传递

概览./d8 --trace-sim hello.js 2>&1 |tee logtracesim.txt

- grep搜索所有的“Call to”和“Return to”的行，就可以得到执行流如何在各个builtins中传递
- 蓝色的部分就是解释器Ignition执行过程

```
CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
Call Builtin InterpreterEntryTrampoline
Call Builtin LdaGlobalHandler
Call Builtin LoadGlobalIC_NoFeedback
Call Builtin LoadIC_NoFeedback
Call Builtin
CEntry_Return1_DontSaveFPRegs_ArgvOnStack_NoBuiltinExit
Call host Runtime::LoadNoFeedbackIC_Miss
Return Builtin LdaGlobalHandler
Call Builtin LdaConstantHandler
Call Builtin CallUndefinedReceiver1Handler
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
Call Builtin HandleApiCall
Call Builtin AdaptorWithBuiltinExitFrame
Call Builtin CEntry_Return1_DontSaveFPRegs_ArgvOnStack_BuiltinExit
Call host Builtin_HandleApiCall
Return Builtin InterpreterEntryTrampoline
Call Builtin ShortStarHandler
Call Builtin ReturnHandler
Return Builtin InterpreterEntryTrampoline
Return Builtin JSEntryTrampoline
Return Builtin JSEntry
```

执行流中ByecodeHandler和Bytecode的对应关系

CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
Call Builtin InterpreterEntryTrampoline
Call Builtin LdaGlobalHandler
Call Builtin LoadGlobalIC_NoFeedback
Call Builtin LoadIC_NoFeedback
Call Builtin
CEntry_Return1_DontSaveFPRegs_ArgvOnStack_NoBuiltinExit
Call host Runtime::LoadNoFeedbackIC_Miss
Return Builtin LdaGlobalHandler
Call Builtin LdaConstantHandler
Call Builtin CallUndefinedReceiver1Handler
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
Call Builtin HandleApiCall
Call Builtin AdaptorWithBuiltinExitFrame
Call Builtin CEntry_Return1_DontSaveFPRegs_ArgvOnStack_BuiltinExit
Call host Builtin_HandleApiCall
Return Builtin InterpreterEntryTrampoline
Call Builtin ShortStarHandler
Call Builtin ReturnHandler
Return Builtin InterpreterEntryTrampoline
Return Builtin JSEntryTrampoline
Return Builtin JSEntry

LdaGlobal [0], [0]
Star1
LdaConstant [1]
Star2
CallUndefinedReceiver1 r1, r2, [2]
Star0
Return

谢 谢

欢迎交流合作

2020/08/15