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-part3

智能软件研究中心 邱吉

qiuji@iscas.ac.cn

2021/10/15

前情回顾

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

- 在上上次课程中，讲述了：
  - hello.js：print(“HelloWorld!”)的字节码和含义
  - 如何从--trace-sim的log文件中，梳理hello.js的解释执行过程
- 在上次课程中，讲述了：
  - d8上hello.js的整体执行流程
  - 如何进入第一部分Prologue部分开始执行
    - Builtin by Builtin

```
CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
Call Builtin InterpreterEntryTrampoline
```

第一部分：Prologue

Hello.js step by step- *HOWTO Interpret in Ignition*

第一部分：Prologue

```
CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
```

第二部分：解释器主体

```
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
```

今天的内容：
Ignition是如何进入
和循环控制的

第三部分：Epilogue

```
Return Builtin JSEntryTrampoline
Return Builtin JSEntry
```

本次内容

- Ignition解释循环主体 InterpreterEntryTrampoline的整体控制流程
- 调试的代码和log： https://github.com/qjivy/v8/tree/v8ignition-learn

复习：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)
```

复习：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

复习：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

第一部分：Prologue

```
CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
```

第二部分：解释器主体

```
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
```

第三部分：Epilogue

```
Return Builtin JSEntryTrampoline
Return Builtin JSEntry
```

InterpreterEntryTrampoline是什么？

- InterpreterEntryTrampoline是解释器的入口、出口（Return）
- 也是Sparkplug/TurboFan里产生的opt code的入口
- 当BytecodeHandler中调用了其他的Builtin和runtime导致无法再返回到BytecodeHandler尾部进行Dispatch的时候，控制流会回到InterpreterEntryTrampoline再次进行下一个字节码的分发
- 代码位置：
  - v8/src/builtins/riscv64/builtins-riscv64.cc
  - ASM类型的builtin，可以去查看相关的Generate_InterpreterEntryTrampoline函数
  - 以及在mksnapshot或者d8启动的时候通过--print-all-code来打印所有的builtins的汇编代码，然后再查看InterpreterEntryTrampoline的

InterpreterEntryTrampoline解析：

入口的参数和接口

```
// The live registers are:
//   o a0 : actual argument count (not including the receiver)
//   o a1: the JS function object being called.
//   o a3: the incoming new target or generator object
//   o cp: our context  ：s7
//   o fp: the caller‘s frame pointer  ：
//   o sp: stack pointer  ：
//   o ra: return address  ：
void Builtins::Generate_InterpreterEntryTrampoline(MacroAssembler* masm) {
  Register closure = a1;
  Register feedback_vector = a2;
  UseScratchRegisterScope temps(masm);
  temps.Include(t0, t1);
  Register scratch = temps.Acquire();
  Register scratch2 = temps.Acquire();
```

InterpreterEntryTrampoline解析：

Step1 从Function object中获得BytecodeArray

```
// Get the bytecode array from the function object and load it into
// kInterpreterBytecodeArrayRegister.（ARCH-specific，for RV64 is t1）
__ LoadTaggedPointerField(
    kScratchReg,
    FieldMemOperand(closure, JSFunction::kSharedFunctionInfoOffset)); //从closure寄存器（指向当前JSFunctionObject，在第二讲中gdb查看过）的偏移JSFunction::kSharedFunctionInfoOffset的位置load一个TaggedPointer到kScratch寄存器中，得到SFI Info object
__ LoadTaggedPointerField(
    kInterpreterBytecodeArrayRegister,
    FieldMemOperand(kScratchReg, SharedFunctionInfo::kFunctionDataOffset));//从SFI Info Object地址偏移SharedFunctionInfo::kFunctionDataOffset的位置load一个TaggedPointer到kInterpreterBytecodeArrayRegister寄存器中，得到FunctionData Object
Label is_baseline;
GetSharedFunctionInfoBytecodeOrBaseline(
    masm, kInterpreterBytecodeArrayRegister, kScratchReg, &is_baseline);//判断FunctionData的类型，如果是BASELINE_DATA_TYPE就跳转到lable is_baseline处进行sparkplug的处理，否则从BytecodeArrayOffset的地方取BytecodeArray
```

分叉一：is_baseline

InterpreterEntryTrampoline解析：

Step2 从Function object中获得BytecodeArray

```
// The bytecode array could have been flushed from the shared function info,
// if so, call into CompileLazy.
Label compile_lazy;
__ GetObjectType(kInterpreterBytecodeArrayRegister, kScratchReg, kScratchReg); //如果刚才load的内容指向的对象并不具有BYTECODE_ARRAY_TYPE，说明这个JSFunction的SFI的FunctionData 域还没有BytecodeArray，需要进行Lazy Compile
__ Branch(&compile_lazy, ne, kScratchReg, Operand(BYTECODE_ARRAY_TYPE));
```

番外：什么是compile_lazy

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

compile_toplevel vs compile_lazy: V8在启动速度和占用内存上的优化措施

lazy.js:

```
function add(a,b) {
return (a+b);
} //function是非toplevel的part，只会进行pre-parse，检查语法错误，并生成SFI，但并不进行compile，不生成Bytecode，直到调用的时候才compile_lazy
add(1,2); // 这是toplevl part，脚本启动时会进行parse和compile_toplevel的动作，生成BytecodeArray
```

print.js:

```
print(”hello”); // 全部都是toplevl part，因此不会引发compile_lazy
```

分叉一：is_baseline
分叉二：compile_lazy

InterpreterEntryTrampoline解析：

Step3 查看FBV的状态，判断是否有优化后的JIT code

```
// Load the feedback vector from the closure.
__ LoadTaggedPointerField(
    feedback_vector,
    FieldMemOperand(closure, JSFunction::kFeedbackCellOffset));
__ LoadTaggedPointerField(
    feedback_vector, FieldMemOperand(feedback_vector, Cell::kValueOffset));
Label push_stack_frame; //这个label是建立解释器栈帧的起始代码
// Check if feedback vector is valid. If valid, check for optimized code
// and update invocation count. Otherwise, setup the stack frame.
__ LoadTaggedPointerField(
    a4, FieldMemOperand(feedback_vector, HeapObject::kMapOffset));
__ Lhu(a4, FieldMemOperand(a4, Map::kInstanceTypeOffset));
__ Branch(&push_stack_frame, ne, a4, Operand(FEEDBACK_VECTOR_TYPE),
          Label::Distance::kNear); //如果不是FBV则确定是解释器模式，跳到push_stack_frame执行，否则执行Optimized的分叉
```

分叉一：is_baseline
分叉二：compile_lazy
分叉三：FBV valid的情况要继续查看optimized状态

InterpreterEntryTrampoline解析：

Step4.1 建立解释器栈帧- push closure，清零flag

```
// Open a frame scope to indicate that there is a frame on the stack.  The
// MANUAL indicates that the scope shouldn't actually generate code to set up
// the frame (that is done below).
__ bind(&push_stack_frame);
FrameScope frame_scope(masm, StackFrame::MANUAL); //Manual类型的Frame完全是手动push/pop
__ PushStandardFrame(closure);//建立Fix header部分
// Reset code age and the OSR arming. The OSR field and BytecodeAgeOffset are
// 8-bit fields next to each other, so we could just optimize by writing a
// 16-bit. These static asserts guard our assumption is valid.
STATIC_ASSERT(BytecodeArray::kBytecodeAgeOffset ==
              BytecodeArray::kOsrNestingLevelOffset + kCharSize);
STATIC_ASSERT(BytecodeArray::kNoAgeBytecodeAge == 0);
__ Sh(zero_reg, FieldMemOperand(kInterpreterBytecodeArrayRegister,
                               BytecodeArray::kOsrNestingLevelOffset)); //函数即将进入解释的流程，说明要么是第一次执行，要么是deopt过了，所以之前的code age和OSR的嵌套层数等profile信息，都应该从BytecodeArray 里面清零
```

番外：解释器栈帧的结构

slot 编号	slot中的对象数据说明		附加说明	
-n	param n	传递的实参n		调用者栈帧
-n+1	param n-1	传递的实参n-1		
	…			
-2	param 1	传递的实参1		
-1	param 0	传递的实参0		
0	return address	返回地址	Fixed Hader	被调用者栈帧
1	privious frame ptr	<- 当前栈帧的fp指向这里		
2	可选的constant pool指针	如果constant pool存在为cp=1，否则cp=0		
2+cp	context	context指针？		
3+cp	JSFunction	SFI指针？		
4+cp	argc	实际传递的参数个数		
5+cp	BytecodeArray	指向BytecodeArray的指针	Interpreter栈帧的专有slot	
6+cp	Bytecodeoffset or Feedback vector	存储了一个Smi（当前在执行的Bytecode的Offset）或者是个指针，指向FBV		
…	…	…	local and temporary variable slot	

v8/src/execution/frame-constants.h

InterpreterEntryTrampoline解析：

Step4.2 建立初始BytecodeArrayOffsetRegister，push BytecodeArrayRegister

```
// Load initial bytecode offset.
__ li(kInterpreterBytecodeOffsetRegister,
      Operand(BytecodeArray::kHeaderSize - kHeapObjectTag));
// Push bytecode array and Smi tagged bytecode array offset.
__ SmiTag(a4, kInterpreterBytecodeOffsetRegister);
__ Push(kInterpreterBytecodeArrayRegister, a4);
```

InterpreterEntryTrampoline解析：

Step4.3 在栈帧上给local和temporary 分配slot，设成undefined

```
// Allocate the local and temporary register file on the stack.
Label stack_overflow;
{
  // Load frame size (word) from the BytecodeArray object.
  __ Lw(a4, FieldMemOperand(kInterpreterBytecodeArrayRegister, BytecodeArray::kFrameSizeOffset)) ; //load FrameSize
  // Do a stack check to ensure we don't go over the limit.
  __ Sub64(a5, sp, Operand(a4));
  __ LoadStackLimit(a2, MacroAssembler::StackLimitKind::kRealStackLimit);
  __ Branch(&stack_overflow, Uless, a5, Operand(a2)); //检查栈溢出
  // If ok, push undefined as the initial value for all register file entries.
  Label loop_header;
  Label loop_check;
  __ LoadRoot(a5, RootIndex::kUndefinedValue); //undefined value常量是从RootIndex::kUndefinedValue来的
  __ BranchShort(&loop_check);
  __ bind(&loop_header); //高亮部分是循环，存储所有的load和temporary到栈上
  // TODO(rmcilroy): Consider doing more than one push per loop iteration.
  __ push(a5);
  // Continue loop if not done.
  __ bind(&loop_check);
  __ Sub64(a4, a4, Operand(kSystemPointerSize));
  __ Branch(&loop_header, ge, a4, Operand(zero_reg));
}
```

分叉一：is_baseline
分叉二：compile_lazy
分叉三：FBV valid的情况要查看optimized状态
分叉四：stack_overflow

InterpreterEntryTrampoline解析：

Step5 设置new_target到BytecodeArray/栈中断检查

```
// If the bytecode array has a valid incoming new target or generator object  register, initialize it with incoming value which was passed in a3.
Label no_incoming_new_target_or_generator_register;
__ Lw(a5, FieldMemOperand(
          kInterpreterBytecodeArrayRegister,
          BytecodeArray::kIncomingNewTargetOrGeneratorRegisterOffset));
__ Branch(&no_incoming_new_target_or_generator_register, eq, a5, Operand(zero_reg), Label::Distance::kNear);
__ CalcScaledAddress(a5, fp, a5, kSystemPointerSizeLog2);
__ Sd(a3, MemOperand(a5));
__ bind(&no_incoming_new_target_or_generator_register);


// Perform interrupt stack check.
// TODO(solanes): Merge with the real stack limit check above.
Label stack_check_interrupt, after_stack_check_interrupt;
__ LoadStackLimit(a5, MacroAssembler::StackLimitKind::kInterruptStackLimit);
__ Branch(&stack_check_interrupt, Uless, sp, Operand(a5),
        Label::Distance::kNear);
__ bind(&after_stack_check_interrupt);
```

分叉一：is_baseline
分叉二：compile_lazy
分叉三：FBV valid的情况要查看optimized状态
分叉四：stack_overflow
分叉五：stack_check_interrupt

InterpreterEntryTrampoline解析：

Step6 正式进入解释执行，do_dispatch

```
 // Load accumulator as undefined.
 __ LoadRoot(kInterpreterAccumulatorRegister, RootIndex::kUndefinedValue);  //先初始化acc reg
// Load the dispatch table into a register and dispatch to the bytecode
 // handler at the current bytecode offset.
 Label do_dispatch；
 __ bind(&do_dispatch);
 __ li(kInterpreterDispatchTableRegister,
      ExternalReference::interpreter_dispatch_table_address(masm->isolate())); //加载Interpreter_dispatch_table的地址到kInterpreterDispatchTableRegister
 __ Add64(a1, kInterpreterBytecodeArrayRegister,
        kInterpreterBytecodeOffsetRegister);
 __ Lbu(a7, MemOperand(a1)); //从BytecodeArray中加载BytecodeOffset指向的Bytecode
 __ CalcScaledAddress(kScratchReg, kInterpreterDispatchTableRegister, a7, kSystemPointerSizeLog2);
 __ Ld(kJavaScriptCallCodeStartRegister, MemOperand(kScratchReg)); //以Bytecode的内容为index，再从Interpreter_dispatch_table中加载解释例程的地址
 __ Call(kJavaScriptCallCodeStartRegister); //跳转到解释例程上去执行
 masm->isolate()->heap()->SetInterpreterEntryReturnPCOffset(masm->pc_offset());
```

番外：分派表是什么？

分派表是将Bytecode的编码与解释例程的地址建立一对一映射的数据结构。在V8中，分派表dispatch_table_是解释器类的私有成员，在解释器初始化时建立。初始化代码位于Interpreter::Initialize函数内。因为前缀码区分了三种同操作数宽度的字节码，因此dispatch_table_共有$3*2^8$个entry。下标范围（0～255）分配给unscaled的8bit操作数字节码，下标范围（256～511）分配给Wide prefix的16字节操作数字节码，下标范围（512～767）分配给ExtraWide prefix的32字节操作数字节码。对于那些固定宽度操作数的字节码来说，它们在256～767范围内的分派表中的表项内容被填充为IllegalHandler。

unscaled index	0	1	2	…	255
	HandlerAddr	HandlerAddr	HandlerAddr	…	
wide index	256	257	258	…	511
extra wide index	512	513	514		767

InterpreterEntryTrampoline解析：

Step7.1 从解释例程返回后，再次do_dispatch或return

```
// Any returns to the entry trampoline are either due to the return bytecode or the interpreter tail calling a builtin and then a dispatch.
// Get bytecode array and bytecode offset from the stack frame. //再次从栈上重建相关寄存器
__ Ld(kInterpreterBytecodeArrayRegister,
      MemOperand(fp, InterpreterFrameConstants::kBytecodeArrayFromFp));
__ Ld(kInterpreterBytecodeOffsetRegister,
      MemOperand(fp, InterpreterFrameConstants::kBytecodeOffsetFromFp));
__ SmiUntag(kInterpreterBytecodeOffsetRegister);
// Either return, or advance to the next bytecode and dispatch.
Label do_return;
__ Add64(a1, kInterpreterBytecodeArrayRegister, kInterpreterBytecodeOffsetRegister);
__ Lbu(a1, MemOperand(a1)); //取下一个字节码
AdvanceBytecodeOffsetOrReturn(masm, kInterpreterBytecodeArrayRegister,
                              kInterpreterBytecodeOffsetRegister, a1, a2, a3,
                              a4, &do_return); //判断是不是return
__ Branch(&do_dispatch);//不是return,就往回跳到do_dispatch
```

InterpreterEntryTrampoline解析：

Step7.2 从解释例程返回后，再次do_dispatch或return

```
__ bind(&do_return);
// The return value is in a0.
LeaveInterpreterFrame(masm, scratch, scratch2); //销毁栈帧，恢复寄存器
__ Jump(ra); //返回
```

Hello.js step by step - *HOWTO InterpreterEntryTrampoline*

第一部分：Prologue

```
CallImpl JSEntry
Call Builtin JSEntryTrampoline
Call Builtin Call_ReceiverIsAny
Call Builtin CallFunction_ReceiverIsAny
```

第二部分：解释器主体

```
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
```

第三部分：Epilogue

```
Return Builtin JSEntryTrampoline
Return Builtin JSEntry
```

流程总结：

1. 检查是否已经被JIT过（baseline/turbofan）
2. 建立解释器栈帧
3. do_dispatch
4. 重入： do_dispatch or return

总结

- 在上上次课程中，讲述了：
  - hello.js：print(“HelloWorld!”)的字节码和含义
  - 如何从--trace-sim的log文件中， 梳理hello.js的解释执行过程
- 在上次课程中，讲述了：
  - d8上hello.js的整体执行流程
  - 如何进入第一部分Prologue部分开始执行
    - Builtin by Builtin

- **本次课程： InterpreterEntryTrampoline的详细流程**
- **掌握技能：后端MacroAssembler code的阅读**

谢 谢

欢迎交流合作

2020/10/15