

《Chrome V8源码》 31.Ignition到底做了什么？（二）



1 摘要

本篇文章是Builtin专题的第六篇，讲解Ignition中的Builtin::kInterpreterEntryTrampoline源码。包括InterpreterEntryTrampoline、Runtime_InterpreterTraceBytecodeEntry和Runtime_InterpreterTraceBytecodeExit源码。

2 InterpreterEntryTrampoline

提示： 本文使用的V8版本是7.9.10，CPU：x64，Builtins-x64.cc，样例代码参见上一篇。

InterpreterEntryTrampoline源码如下：

```
1. void Builtins::Generate_InterpreterEntryTrampoline(MacroAssembler* masm) {
2.   Register closure = rdi;
3.   Register feedback_vector = rbx;
4.   __ LoadTaggedPointerField(
5.     rax, FieldOperand(closure, JSFunction::kSharedFunctionInfoOffset));
6.   __ LoadTaggedPointerField(
7.     kInterpreterBytecodeArrayRegister,
8.     FieldOperand(rax, SharedFunctionInfo::kFunctionDataOffset));
9.   GetSharedFunctionInfoBytecode(masm, kInterpreterBytecodeArrayRegister,
10.    kScratchRegister);
11.   Label compile_lazy;
12.   __ CmpObjectType(kInterpreterBytecodeArrayRegister, BYTECODE_ARRAY_TYPE,
    rax);
```

```

13.  __ j(not_equal, &compile_lazy);
14.  __ bind(&push_stack_frame);
15.  FrameScope frame_scope(masm, StackFrame::MANUAL);
16.  __ pushq(rbp); // Caller's frame pointer.
17.  __ movq(rbp, rsp);
18.  __ Push(rsi); // Callee's context.
19.  __ Push(rdi); // Callee's JS function.
20.  __ movw(FieldOperand(kInterpreterBytecodeArrayRegister,
21.                      BytecodeArray::kOsrNestingLevel0Offset),
22.          Immediate(0));
23.  __ movq(kInterpreterBytecodeOffsetRegister,
24.          Immediate(BytecodeArray::kHeaderSize - kHeapObjectTag));
25.  __ Push(kInterpreterBytecodeArrayRegister);
26.  __ SmiTag(rcx, kInterpreterBytecodeOffsetRegister);
27.  __ Push(rcx);
28.  {
29.    //Allocate the local and temporary register file on the stack.
30.    //省略.....
31.  }
32.  __ LoadRoot(kInterpreterAccumulatorRegister, RootIndex::kUndefinedValue);
33.  Label do_dispatch;
34.  __ bind(&do_dispatch);
35.  __ Move(
36.      kInterpreterDispatchTableRegister,
37.      ExternalReference::interpreter_dispatch_table_address(masm-
>isolate()));
38.  __ movzxbq(r11, Operand(kInterpreterBytecodeArrayRegister,
39.                          kInterpreterBytecodeOffsetRegister, times_1, 0));
40.  __ movq(kJavaScriptCallCodeStartRegister,
41.          Operand(kInterpreterDispatchTableRegister, r11,
42.                  times_system_pointer_size, 0));
43.  __ call(kJavaScriptCallCodeStartRegister);
44.  masm->isolate()->heap()->SetInterpreterEntryReturnPCOffset(masm-
>pc_offset());
45.  __ movq(kInterpreterBytecodeArrayRegister,
46.          Operand(rbp, InterpreterFrameConstants::kBytecodeArrayFromFp));
47.  __ movq(kInterpreterBytecodeOffsetRegister,
48.          Operand(rbp, InterpreterFrameConstants::kBytecodeOffsetFromFp));
49.  __ SmiUntag(kInterpreterBytecodeOffsetRegister,
50.             kInterpreterBytecodeOffsetRegister);
51.  Label do_return;
52.  __ movzxbq(rbx, Operand(kInterpreterBytecodeArrayRegister,
53.                          kInterpreterBytecodeOffsetRegister, times_1, 0));
54.  AdvanceBytecodeOffsetOrReturn(masm, kInterpreterBytecodeArrayRegister,
55.                                kInterpreterBytecodeOffsetRegister, rbx, rcx,
56.                                &do_return);
57.  __ jmp(&do_dispatch);
58.  __ bind(&do_return);
59.  LeaveInterpreterFrame(masm, rbx, rcx);
60.  __ ret(0);
61.  __ bind(&compile_lazy);
62.  GenerateTailCallToReturnedCode(masm, Runtime::kCompileLazy);
63.  __ int3();
64.  }

```

上述代码中，第4行代码：从JSFunction中取出SharedFunction并存储到rax中；第6行代码：从SharedFunctionInfo获取kFunctionDataOffset的数据并存储到kInterpreterBytecodeArrayRegister中；第9行代码：加载Bytecodearray到kInterpreterBytecodeArrayRegister。

细节说明：

(1) `FieldOperand(x,y)`方法中x是基址，y是偏移量，该方法用于返回x+y的位置的数据；
(2) 因为SharedFunction::kFunctionDataOffset可能存储Bytecodearray或Builtin，所以执行完第6行代码后需要用第9行代码判断kInterpreterBytecodeArrayRegister中的数据是否是Bytecodearray。
上述第10-13行代码：判断kInterpreterBytecodeArrayRegister的值是Bytecodearray还是Builtins::kCompileLazy，根据判断结果跳转到相应的Label；第15-19行代码存储caller的栈帧并把callee的信息压入堆栈。第20-27行代码获取Bytecodearray中第一条Bytecode的偏移量并压入堆栈。BytecodeArray类继承自FixedArrayBase，FixedArrayBase又继承自HeapObject，所以获取第一条Bytecode时需要使用刚刚获取的偏移量。
上述第32行初始化kInterpreterAccumulatorRegister；第35行代码加载dispatch到kInterpreterDispatchTableRegister；第38-40行代码加载第一条Bytecode到kJavaScriptCallCodeStartRegister；第43行代码开始执行Bytecode。所有Bytecode都执行完成后会跳转到第44行代码以设置返回地址。
两种情况下会执行上述第45-63行代码，(1) 当全部Bytecode执行完后，Bytecode的结尾会调用Dispatch()，所以只有全部执行完时才会返回；(2) 在Bytecode执行过程中调用了其它Builtin，因为调用其它Builtin要重新构建堆栈，所以还要用InterpreterEntryTrampoline。
至此，InterpreterEntryTrampoline分析完毕。

3 Register

InterpreterEntryTrampoline中使用了很多Register，列表如下：

```
constexpr Register kReturnRegister0 = rax;
constexpr Register kReturnRegister1 = rdx;
constexpr Register kReturnRegister2 = r8;
constexpr Register kJSFunctionRegister = rdi;
constexpr Register kContextRegister = rsi;
constexpr Register kAllocateSizeRegister = rdx;
constexpr Register kSpeculationPoisonRegister = r12;
constexpr Register kInterpreterAccumulatorRegister = rax;
constexpr Register kInterpreterBytecodeOffsetRegister = r9;
constexpr Register kInterpreterBytecodeArrayRegister = r14;
constexpr Register kInterpreterDispatchTableRegister = r15;
//省略.....
```

在InterpreterEntryTrampoline中常用到的寄存器是rax、rdi、rdx和r15，其中被多次提及的r15负责Bytecode的调度。我在汇编中调试Bytecode时，r15寄存器常被用作“入口标记”，即看到r15就说明一条Bytecode开始了，再次看到r15就说明这条Bytecode结束了。

4 InterpreterTraceBytecodeEntry和InterpreterTraceBytecodeExit

这两个方法用于跟踪Bytecode的解释过程，`InterpreterTraceBytecodeEntry`可以查看寄存器状态；Bytecode执行后调用`InterpreterTraceBytecodeExit`。源码如下：

```

1.  RUNTIME_FUNCTION(Runtime_InterpreterTraceBytecodeEntry) {
2.      if (!FLAG_trace_ignition) {
3.          return ReadOnlyRoots(isolate).undefined_value();
4.      }
5.      SealHandleScope shs(isolate);
6.      DCHECK_EQ(3, args.length());
7.      CONVERT_ARG_HANDLE_CHECKED(BytecodeArray, bytecode_array, 0);
8.      CONVERT_SMI_ARG_CHECKED(bytecode_offset, 1);
9.      CONVERT_ARG_HANDLE_CHECKED(Object, accumulator, 2);
10.     int offset = bytecode_offset - BytecodeArray::kHeaderSize + kHeapObjectTag;
11.     interpreter::BytecodeArrayIterator bytecode_iterator(bytecode_array);
12.     AdvanceToOffsetForTracing(bytecode_iterator, offset);
13.     if (offset == bytecode_iterator.current_offset()) {
14.         StdoutStream os;
15.         // Print bytecode.
16.         const uint8_t* base_address = reinterpret_cast<const uint8_t*>(
17.             bytecode_array->GetFirstBytecodeAddress());
18.         const uint8_t* bytecode_address = base_address + offset;
19.         os << " -> " << static_cast<const void*>(bytecode_address) << " @ "
20.             << std::setw(4) << offset << " : ";
21.         interpreter::BytecodeDecoder::Decode(os, bytecode_address,
22.             bytecode_array->parameter_count());
23.         os << std::endl;
24.         // Print all input registers and accumulator.
25.         PrintRegisters(isolate, os, true, bytecode_iterator, accumulator);
26.         os << std::flush;
27.     }
28.     return ReadOnlyRoots(isolate).undefined_value();
29. }
30. //分隔线.....
31. RUNTIME_FUNCTION(Runtime_InterpreterTraceBytecodeExit) {
32.     if (!FLAG_trace_ignition) {
33.         return ReadOnlyRoots(isolate).undefined_value();
34.     }
35.     SealHandleScope shs(isolate);
36.     DCHECK_EQ(3, args.length());
37.     CONVERT_ARG_HANDLE_CHECKED(BytecodeArray, bytecode_array, 0);
38.     CONVERT_SMI_ARG_CHECKED(bytecode_offset, 1);
39.     CONVERT_ARG_HANDLE_CHECKED(Object, accumulator, 2);
40.     int offset = bytecode_offset - BytecodeArray::kHeaderSize + kHeapObjectTag;
41.     interpreter::BytecodeArrayIterator bytecode_iterator(bytecode_array);
42.     AdvanceToOffsetForTracing(bytecode_iterator, offset);
43.     if (bytecode_iterator.current_operand_scale() ==
44.         interpreter::OperandScale::kSingle ||
45.         offset > bytecode_iterator.current_offset()) {
46.         StdoutStream os;
47.         // Print all output registers and accumulator.
48.         PrintRegisters(isolate, os, false, bytecode_iterator, accumulator);
49.         os << std::flush;

```

```

50.     }
51.     return ReadOnlyRoots(isolate).undefined_value();
52. }

```

Bytecode执行前后会分别调用上述两个方法，但需要把FLAG_trace_ignition（第2行代码）的值设置为True,其声明在flags-definitions.h中，具体位置是DEFINE_BOOL(trace_ignition, false, "trace the bytecodes executed by the ignition interpreter")。第21行代码输出Bytecode到终端，阅读BytecodeDecoder::Decode()源码可以看明白Bytecode和operand的编码方式，这有助于理解dispatch和JS调用堆栈。

下面给出PrintRegisters源码：

```

1. void PrintRegisters(Isolate* isolate, std::ostream& os, bool is_input,
2.                     interpreter::BytecodeArrayIterator&
3.                     bytecode_iterator, // NOLINT(runtime/references)
4.                     Handle<Object> accumulator) {
5.     interpreter::Bytecode bytecode = bytecode_iterator.current_bytecode();
6.     // Print accumulator.
7.     if ((is_input && interpreter::Bytecodes::ReadsAccumulator(bytecode)) ||
8.         (!is_input && interpreter::Bytecodes::WritesAccumulator(bytecode))) {
9.         os << "      [ " << kAccumulator << kArrowDirection;
10.        accumulator->ShortPrint();
11.        os << " ]" << std::endl;
12.    }
13.    // Print the registers.
14.    JavaScriptFrameIterator frame_iterator(isolate);
15.    InterpretedFrame* frame =
16.        reinterpret_cast<InterpretedFrame*>(frame_iterator.frame());
17.    int operand_count = interpreter::Bytecodes::NumberOfOperands(bytecode);
18.    for (int operand_index = 0; operand_index < operand_count; operand_index++)
19.    {
20.        interpreter::OperandType operand_type =
21.            interpreter::Bytecodes::GetOperandType(bytecode, operand_index);
22.        bool should_print =
23.            is_input
24.            ?
25.            interpreter::Bytecodes::IsRegisterInputOperandType(operand_type)
26.            :
27.            interpreter::Bytecodes::IsRegisterOutputOperandType(operand_type);
28.        if (should_print) {
29.            interpreter::Register first_reg =
30.                bytecode_iterator.GetRegisterOperand(operand_index);
31.            int range = bytecode_iterator.GetRegisterOperandRange(operand_index);
32.            for (int reg_index = first_reg.index();
33.                 reg_index < first_reg.index() + range; reg_index++) {
34.                Object reg_object = frame->ReadInterpreterRegister(reg_index);
35.                os << "      [ " << std::setw(kRegFieldWidth)
36.                    << interpreter::Register(reg_index).ToString(
37.                        bytecode_iterator.bytecode_array()->parameter_count())
38.                    << kArrowDirection;
39.                reg_object.ShortPrint(os);
40.                os << " ]" << std::endl;

```

```
38.     }  
39.     }  
40.   }  
41. }
```

上述第14-17行代码计算操作数的数量。第20-37行代码输出寄存器的值。通过阅读`PrintRegisters()`方法，我们可以学到三个有用的知识点：

- (1) 读取寄存器的方法；
- (2) V8中打印数据的方法；
- (3) `InterpretedFrame`的数据结构。

这三点可以帮助我们更好地了解Bytecode的执行过程。**提示**V8中功能全面的打印方法是`logger`。

技术总结

(1) `SharedFunction::kFunctionDataOffset`位置存储的内容可能是`Bytecodearray`也可能是`Builtins::kCompileLazy`；

(2) `BytecodeDecoder::Decode()`和`PrintRegisters()`很重要，可以帮助我们理解Bytecode的执行过程。

好了，今天到这里，下次见。

个人能力有限，有不足与纰漏，欢迎批评指正

微信：qq9123013 备注：v8交流 邮箱：v8blink@outlook.com

本文由灰豆原创发布

转载出处：<https://www.anquanke.com/post/id/261687>

安全客 - 有思想的安全新媒体