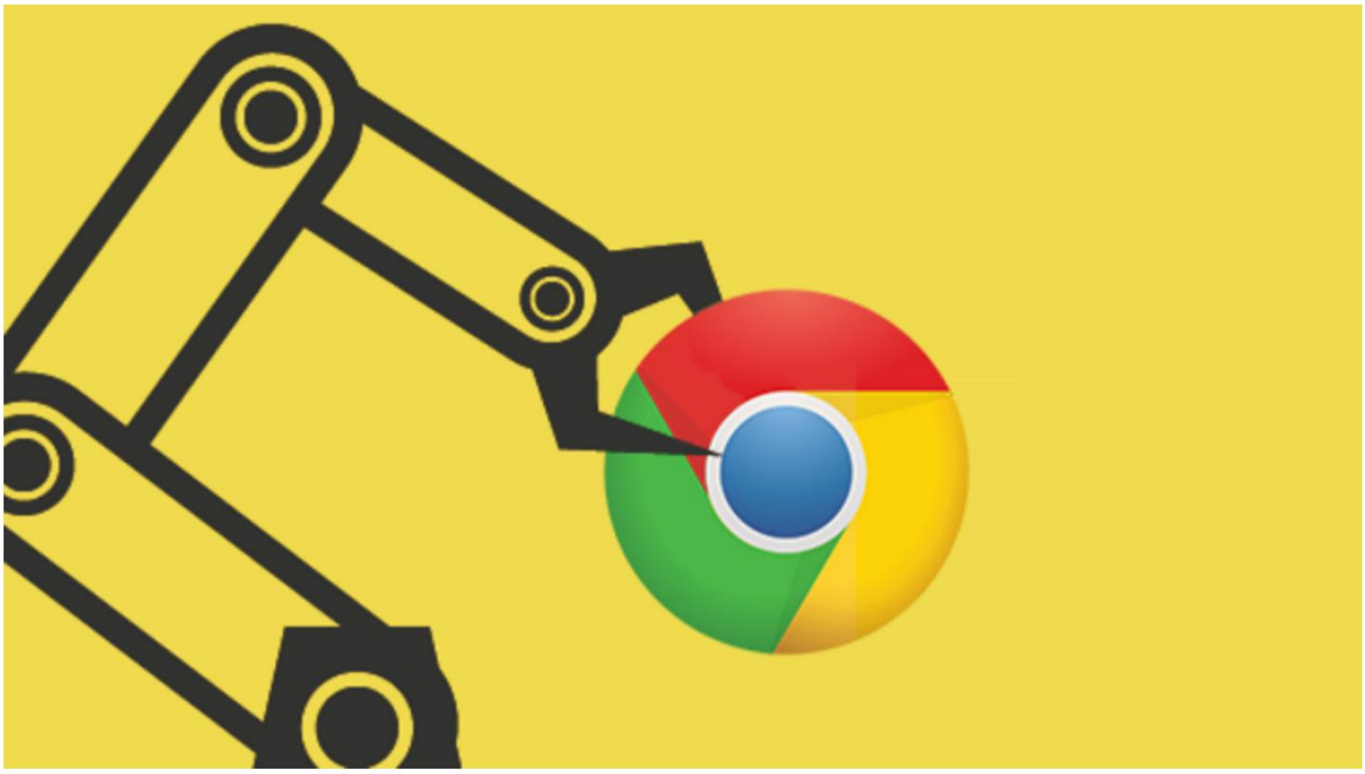
"Chrome V8 Source Code" 31.What exactly does Ignition do? (two)



1 abstract

This article is the sixth of the Builtin topic, explaining the Builtin::kInterpreterEntryTrampoline source code in Ignition. include InterpreterEntryTrampoline, Runtime_InterpreterTraceBytecodeEntry and Runtime_InterpreterTraceBytecodeExit Source code.

2 InterpreterEntryTrampoline

Tip: The V8 version used in this article is 7.9.10, CPU: x64, Builtins-x64.cc, see the previous article for sample code.

The source code of InterpreterEntryTrampoline is as follows:

```
void Builtins::Generate_InterpreterEntryTrampoline(MacroAssembler* masm) {
Register closure = rdi;
Register feedback_vector = rbx;
    __LoadTaggedPointerField (
          rax, FieldOperand(closure, JSFunction::kSharedFunctionInfoOffset));
    __LoadTaggedPointerField (
          kInterpreterBytecodeArrayRegister,
          FieldOperand(rax, SharedFunctionInfo::kFunctionDataOffset));
GetSharedFunctionInfoBytecode(masm, kInterpreterBytecodeArrayRegister,
kScratchRegister );
       Label compile_lazy;
       __CmpObjectType (kInterpreterBytecodeArrayRegister, BYTECODE_ARRAY_TYPE,
rax);
```

```
    __ j(not_equal, &compile_lazy);
    __ bind(&push_stack_frame);
    FrameScope frame_scope(masm, StackFrame::MANUAL);
    __ pushq(rbp); // Caller's frame pointer.
    __ movq(rbp, rsp);
    __ Push(rsi); // Callee's context.
    __ Push(rdi); // Callee's JS function.
    __ movw(FieldOperand(kInterpreterBytecodeArrayRegister,
                         BytecodeArray::kOsrNestingLevelOffset),
            Immediate(0));
    __ movq(kInterpreterBytecodeOffsetRegister,
            Immediate(BytecodeArray::kHeaderSize - kHeapObjectTag));
    __Push (kInterpreterBytecodeArrayRegister);
    __SmiTag (rcx, kInterpreterBytecodeOffsetRegister);
    __Push (rcx);
    {
      //Allocate the local and temporary register file on the stack.
      //Omit.......................
    }
    __ LoadRoot(kInterpreterAccumulatorRegister, RootIndex::kUndefinedValue);
    Label do_dispatch;
    __ bind(&do_dispatch);
    __Move (
        kInterpreterDispatchTableRegister,
        ExternalReference::interpreter_dispatch_table_address(masm->isolate()));
    __ movzxbq(r11, Operand(kInterpreterBytecodeArrayRegister,
                            kInterpreterBytecodeOffsetRegister, times_1, 0));
    __ movq(kJavaScriptCallCodeStartRegister,
            Operand(kInterpreterDispatchTableRegister, r11,
                    times_system_pointer_size, 0));
    __ call(kJavaScriptCallCodeStartRegister);
    masm->isolate()->heap()->SetInterpreterEntryReturnPCOffset(masm->pc_offset());
    __ movq(kInterpreterBytecodeArrayRegister,
            Operand(rbp, InterpreterFrameConstants::kBytecodeArrayFromFp));
    __ movq(kInterpreterBytecodeOffsetRegister,
            Operand(rbp, InterpreterFrameConstants::kBytecodeOffsetFromFp));
    __SmiUntag (kInterpreterBytecodeOffsetRegister,
                kInterpreterBytecodeOffsetRegister);
    Label do_return;
    __ movzxbq(rbx, Operand(kInterpreterBytecodeArrayRegister,
                            kInterpreterBytecodeOffsetRegister, times_1, 0));
    AdvanceBytecodeOffsetOrReturn(masm, kInterpreterBytecodeArrayRegister,
                                  kInterpreterBytecodeOffsetRegister, rbx, rcx,
                                  &do_return);
    __jmp (&do_dispatch);
    __ bind(&do_return);
    LeaveInterpreterFrame(masm, rbx, rcx);
    __ret (0);
    __ bind(&compile_lazy);
    GenerateTailCallToReturnedCode(masm, Runtime::kCompileLazy);
      __ int3();
  }
```

In the above code, the 4th line of code: remove the SharedFunction from JSFunction and store it in rax; the 6th line of code: get the kFunctionDataOffset data from SharedFunctionInfo and store it in kInterpreterBytecodeArrayRegister; the 9th line of code: load the Bytecodearray to kInterpreterBytecodeArrayRegister. **Detailed description: (1)** In the FieldOperand(x,y) method, x is the base address and y is the offset. This method is used to return the data at the position of x+y; **(2)** Because SharedFunction::kFunctionDataOffset may store Bytecodearray or Builtin, so after executing the 6th line of code, you need to use the 9th line of code to determine whether the data in kInterpreterBytecodeArrayRegister is a Bytecodearray. Lines 10-13 of the above code: Determine whether the value of kInterpreterBytecodeArrayRegister is Bytecodarray or Builtins::kCompileLazy, and jump to the corresponding Label based on the judgment result; Lines 15-19 of the code store the caller's stack frame and push the callee's information onto the stack. Lines 20-27 of code obtain the offset of the first Bytecode in the Bytecodearray and push it onto the stack. The BytecodeArray class inherits from FixedArrayBase, and FixedArrayBase inherits from HeapObject, so when obtaining the first Bytecode, you need to use the offset just obtained. Line 32 above initializes kInterpreterAccumulatorRegister; line 35 of code loads dispatch to kInterpreterDispatchTableRegister; lines 38-40 of code loads the first Bytecode to kJavaScriptCallCodeStartRegister; line 43 of code starts executing Bytecode. After all Bytecode is executed, it will jump to line 44 of code to set the return address. In two cases, the above lines 45-63 of the code will be executed. (1) When all Bytecode is executed, Dispatch() will be called at the end of Bytecode, so it will only return when all Bytecode is executed; (2) During the execution of Bytecode Other Builtins are called in. Because calling other Builtins requires rebuilding the stack, InterpreterEntryTrampoline is also used. At this point, the analysis of InterpreterEntryTrampoline is completed.

3 Register

There are many Registers used in InterpreterEntryTrampoline, the list is as follows:

```
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 kInterpreterAccumulatorReg
ister = rax; constexpr Register kInterpreterBytecodeOffsetRegister = r9; constexpr
Register kInterpreterBytecodeArrayRegister = r14; constexpr Register
kInterpreterDispatchTableRegister = r15; //Omitted............
```

The registers commonly used in InterpreterEntryTrampoline are rax, rdi, rdx and r15, among which r15, which has been mentioned many times, is responsible for Bytecode scheduling. When I debug Byteocde in assembly, the r15 register is often used as an "entry mark", that is, seeing r15 means something Bytecode has started, and seeing r15 again means that this Bytecode has ended.

4 InterpreterTraceBytecodeEntry and InterpreterTraceBytecodeExit

These two methods are used to track the interpretation process of Bytecode. InterpreterTraceBytecodeEntry can view the register status; InterpreterTraceBytecodeExit is called after Bytecode is executed. The source code is as follows:

```
RUNTIME_FUNCTION(Runtime_InterpreterTraceBytecodeEntry) {
    if (!FLAG_trace_ignition) {
      return ReadOnlyRoots(isolate).undefined_value();
    }
    SealHandleScope shs(isolate);
    DCHECK_EQ(3, args.length());
    CONVERT_ARG_HANDLE_CHECKED(BytecodeArray, bytecode_array, 0);
    CONVERT_SMI_ARG_CHECKED(bytecode_offset, 1);
    CONVERT_ARG_HANDLE_CHECKED(Object, accumulator, 2);
    int offset = bytecode_offset - BytecodeArray::kHeaderSize + kHeapObjectTag;
    interpreter::BytecodeArrayIterator bytecode_iterator(bytecode_array);
    AdvanceToOffsetForTracing(bytecode_iterator, offset);
    if (offset == bytecode_iterator.current_offset()) {
      StdoutStream os;
      // Print bytecode.
      const uint8_t* base_address = reinterpret_cast<const uint8_t*>(
          bytecode_array->GetFirstBytecodeAddress());
      const uint8_t* bytecode_address = base_address + offset;
      os << " -> " << static_cast<const void*>(bytecode_address) << : ";   " @ "
         << std::setw(4) << offset <<
      interpreter::BytecodeDecoder::Decode(os, bytecode_address,
                                           bytecode_array->parameter_count());
      os << std::endl;
      // Print all input registers and accumulator.
      PrintRegisters(isolate, os, true, bytecode_iterator, accumulator);
      os << std::flush;
    }
    return ReadOnlyRoots(isolate).undefined_value();
}
//Separator line............................
RUNTIME_FUNCTION(Runtime_InterpreterTraceBytecodeExit) {
    if (!FLAG_trace_ignition) {
      return ReadOnlyRoots(isolate).undefined_value();
    }
    SealHandleScope shs(isolate);
    DCHECK_EQ(3, args.length());
    CONVERT_ARG_HANDLE_CHECKED(BytecodeArray, bytecode_array, 0);
    CONVERT_SMI_ARG_CHECKED(bytecode_offset, 1);
    CONVERT_ARG_HANDLE_CHECKED(Object, accumulator, 2);
    int offset = bytecode_offset - BytecodeArray::kHeaderSize + kHeapObjectTag;
    interpreter::BytecodeArrayIterator bytecode_iterator(bytecode_array);
    AdvanceToOffsetForTracing(bytecode_iterator, offset);
    if (bytecode_iterator.current_operand_scale() ==
            interpreter::OperandScale::kSingle ||
        offset > bytecode_iterator.current_offset()) {
      StdoutStream os;
      // Print all output registers and accumulator.
      PrintRegisters(isolate, os, false, bytecode_iterator, accumulator);
      os << std::flush;
    }
    return ReadOnlyRoots(isolate).undefined_value();
}
```

The above two methods will be called before and after Bytecode is executed, but the value of FLAG_trace_ignition (line 2 of code) needs to be set to True. It is clearly in flags-definitions.h, the specific location is DEFINE_BOOL(trace_ignition, false,"trace the bytecodes executed by the ignition interpreter"). Line 21 of the code outputs Bytecode to the terminal, read BytecodeDecoder::Decode() source code can understand the encoding methods of Bytecode and operand, which is helpful for understanding dispatch and JS debugging. Use stack.

The PrintRegisters source code is given below:

```
void PrintRegisters(Isolate* isolate, std::ostream& os, bool is_input,
                    interpreter::BytecodeArrayIterator&
                        bytecode_iterator, // NOLINT(runtime/references)
                    Handle<Object> accumulator) {
  interpreter::Bytecode bytecode = bytecode_iterator.current_bytecode();
  // Print accumulator.
  if ((is_input && interpreter::Bytecodes::ReadsAccumulator(bytecode)) ||
      (!is_input && interpreter::Bytecodes::WritesAccumulator(bytecode))) {
    os << "                [ " << kAccumulator << kArrowDirection;
    accumulator->ShortPrint();
    os << " ]" << std::endl;
  }
  // Print the registers.
  JavaScriptFrameIterator frame_iterator(isolate);
  InterpretedFrame* frame =
      reinterpret_cast<InterpretedFrame*>(frame_iterator.frame());
  int operand_count = interpreter::Bytecodes::NumberOfOperands(bytecode);
  for (int operand_index = 0; operand_index < operand_count; operand_index++)
{
    interpreter::OperandType operand_type =
        interpreter::Bytecodes::GetOperandType(bytecode, operand_index);
    bool should_print =
        is_input
            ?
interpreter::Bytecodes::IsRegisterInputOperandType(operand_type)
            :
interpreter::Bytecodes::IsRegisterOutputOperandType(operand_type);
    if (should_print) {
      interpreter::Register first_reg =
          bytecode_iterator.GetRegisterOperand(operand_index);
      int range = bytecode_iterator.GetRegisterOperandRange(operand_index);
      for (int reg_index = first_reg.index();
           reg_index < first_reg.index() + range; reg_index++) {
        Object reg_object = frame->ReadInterpreterRegister(reg_index);
        os << "                [ " << std::setw(kRegFieldWidth)
           << interpreter::Register(reg_index).ToString(
                  bytecode_iterator.bytecode_array()->parameter_count())
           << kArrowDirection;
        reg_object.ShortPrint(os);
        os << " ]" << std::endl;
```

```
                }
            }
        }
}
```

Lines 14-17 above calculate the number of operands. Lines 20-37 code output the value of the register. By reading the PrintRegisters() method, we can learn three useful knowledge points:

(1) How to read registers; (2)
How to print data in V8; (3) The
data structure of InterpretedFrame. These

three points can help us better understand the execution process of Bytecode. Tip: The most comprehensive printing method in V8 is logger.

Technical

summary (1) The content stored in the location of SharedFunction::kFunctionDataOffset may be Bytecodearray or Builtins::kCompileLazy;

(2) BytecodeDecoder::Decode() and PrintRegisters() are very important and can help us understand the execution process of Bytecode.

Okay, that's it for today, see you next time.

Personal abilities are limited, there are shortcomings and mistakes,

criticisms and corrections are welcome WeChat: qq9123013 Note: v8 communication email: v8blink@outlook.com