《Chrome V8源码》30.Ignition到底做了什么?

1 摘要

本篇文章是Builtin专题的第五篇，讲解Ignition解释Bytecode之前需要做的准备工作，这些工作由一系列Builtin共同完成，工作内容包括：构建堆栈、压入参数等。本文通过分析这一系列Builtin的工作流程，来了解Ignition为Bytecode做了哪些准备工作。

2 Ignition概述

提示： 本文使用的V8版本是7.9.10，CPU：x64，Builtins-x64.cc
测试样例如下：

```
function ignition(s) {
    this.slogan=s;
    this.start=function(){eval('console.log(this.slogan);')}
}
worker = new ignition("here we go!");
worker.start();
```

从`V8_WARN_UNUSED_RESULT MaybeHandle<Object> Invoke(Isolate* isolate,const InvokeParams& params)`讲起，因为Ignition即将从这里开始工作，源码如下：

```
1.  InvokeParams InvokeParams::SetUpForCall(Isolate* isolate,
2.                                          Handle<Object> callable,
3.                                          Handle<Object> receiver, int argc,
4.                                          Handle<Object>* argv) {
5.    InvokeParams params;
6.    params.target = callable;
7.    params.receiver = NormalizeReceiver(isolate, receiver);
8.    params.argc = argc;
9.    params.argv = argv;
10.    params.new_target = isolate->factory()->undefined_value();
11.  //省略.................
12.    return params;
13.  }
14.  //..............分隔线.............................
15.  V8_WARN_UNUSED_RESULT MaybeHandle<Object> Invoke(Isolate* isolate,
16.                                                   const InvokeParams& params)
{
17.    Handle<Code> code =
18.        JSEntry(isolate, params.execution_target, params.is_construct);
19.    {
20.      if (params.execution_target == Execution::Target::kCallable) {
21.        using JSEntryFunction = GeneratedCode<Address(
22.            Address root_register_value, Address new_target, Address target,
23.            Address receiver, intptr_t argc, Address** argv)>;
24.         JSEntryFunction stub_entry =
25.             JSEntryFunction::FromAddress(isolate, code->InstructionStart());
26.         Address orig_func = params.new_target->ptr();
27.         Address func = params.target->ptr();
28.         Address recv = params.receiver->ptr();
29.         Address** argv = reinterpret_cast<Address**>(params.argv);
30.         RuntimeCallTimerScope timer(isolate,
RuntimeCallCounterId::kJS_Execution);
31.         value = Object(stub_entry.Call(isolate->isolate_data()-
>isolate_root(),
32.                                        orig_func, func, recv, params.argc,
argv));
33.       } else {
34.       }
35.     }
36.   }
```

上述代码的重点内容如下：

(1) 第1-13行代码SetUpForCall()封装参数params，第6行代码target是测试代码的JSFunction，第8，9行代码的值是0和nullptr;

(2) 第17行代码code的值是Builtin::kJSEntry的入口地址；

(3) 第24行代码stub_entry的值是Builtin::kJSEntry的第一条指令位置;

(4) 第27行代码func是测试样例的JSFunction；

(5) 第21行代码stub_entry.Call声明，stub_entry.Call的第3个参数target是func,即测试代码的JSFunction；

(6) 第31行代码stub_entry.Call开始执行Builtin::kJSEntry。

进入Builtin::kJSEntry之前，先说明几个Ignition要用到的JSFunction成员：

（1） SharedFunction成员，该成员存储JSFunciton对应的SharedFunction实例，该成员的位置偏移是kSharedFunctionInfoOffset；
（2） code成员，该成员存储Builtins::kInterpreterEntryTrampoline，该成员的位置偏移是kCodeOffset；
（3） context成员，该成员存储本次使用的context，该成员的位置偏移是kContextOffset。
接下来从Builtin::kJSEntry开始说起。

3 Builtin::kJSEntry

```
1.  void Builtins::Generate_JSEntry(MacroAssembler* masm) {
2.    Generate_JSEntryVariant(masm, StackFrame::ENTRY,
3.                            Builtins::kJSEntryTrampoline);
4.  }
5.  //...........分隔线.................
6.  void Generate_JSEntryVariant(MacroAssembler* masm, StackFrame::Type type,
7.                               Builtins::Name entry_trampoline) {
8.    Label invoke, handler_entry, exit;
9.    Label not_outermost_js, not_outermost_js_2;
10.    {  // NOLINT. Scope block confuses linter.
11.      NoRootArrayScope uninitialized_root_register(masm);
12.  //..............省略...........
13.      // Initialize the root register.
14.      // C calling convention. The first argument is passed in arg_reg_1.
15.      __ movq(kRootRegister, arg_reg_1);
16.    // Save copies of the top frame descriptor on the stack.
17.    ExternalReference c_entry_fp = ExternalReference::Create(
18.        IsolateAddressId::kCEntryFPAddress, masm->isolate());
19.    {
20.      Operand c_entry_fp_operand = masm-
>ExternalReferenceAsOperand(c_entry_fp);
21.      __ Push(c_entry_fp_operand);
22.    }
23.    // Store the context address in the previously-reserved slot.
24.    ExternalReference context_address = ExternalReference::Create(
25.        IsolateAddressId::kContextAddress, masm->isolate());
26.    __ Load(kScratchRegister, context_address);
27.    static constexpr int kOffsetToContextSlot = -2 * kSystemPointerSize;
28.    __ movq(Operand(rbp, kOffsetToContextSlot), kScratchRegister);
29.    __ jmp(&invoke);
30.    __ bind(&handler_entry);
31.    // Store the current pc as the handler offset. It's used later to create
the
32.    // handler table.
33.    masm->isolate()->builtins()->SetJSEntryHandlerOffset(handler_entry.pos());
34.    // Caught exception: Store result (exception) in the pending exception
35.    // field in the JSEnv and return a failure sentinel.
36.    ExternalReference pending_exception = ExternalReference::Create(
37.        IsolateAddressId::kPendingExceptionAddress, masm->isolate());
38.    __ Store(pending_exception, rax);
39.    __ LoadRoot(rax, RootIndex::kException);
40.    __ jmp(&exit);
```

```
41.    // Invoke: Link this frame into the handler chain.
42.    __ bind(&invoke);
43.    __ PushStackHandler();
44.    // Invoke the function by calling through JS entry trampoline builtin and
45.    // pop the faked function when we return.
46.    Handle<Code> trampoline_code =
47.        masm->isolate()->builtins()->builtin_handle(entry_trampoline);
48.    __ Call(trampoline_code, RelocInfo::CODE_TARGET);
49.  //.............省略.............
50.  }
```

上述代码第2行进入Generate_JSEntryVariant()方法，该方法的第3个参数Builtins::kJSEntryTrampoline在第47行会被用到。第15行代码arg_reg_1的值是isolate->isolate_data()->isolate_root()。第17-22行代码把当前的top frame压入堆栈。第24-26加载context到ScratchRegister。第36-39行设置异常处理。第46-48行代码调用Builtins::kJSEntryTrampoline。

4 Builtins::kJSEntryTrampoline

```
1.  void Builtins::Generate_JSEntryTrampoline(MacroAssembler* masm) {
2.    Generate_JSEntryTrampolineHelper(masm, false);
3.  }
4.  //.............分隔线.................
5.  static void Generate_JSEntryTrampolineHelper(MacroAssembler* masm,
6.                                                bool is_construct) {
7.    // Expects six C++ function parameters.
8.    // - Address root_register_value
9.    // - Address new_target (tagged Object pointer)
10.    // - Address function (tagged JSFunction pointer)
11.    // - Address receiver (tagged Object pointer)
12.    // - intptr_t argc
13.    // - Address** argv (pointer to array of tagged Object pointers)
14.    // (see Handle::Invoke in execution.cc).
15.    // Open a C++ scope for the FrameScope.
16.    {
17.      // Platform specific argument handling. After this, the stack contains
18.      // an internal frame and the pushed function and receiver, and
19.      // register rax and rbx holds the argument count and argument array,
20.      // while rdi holds the function pointer, rsi the context, and rdx the
21.      // new.target.
22.      // MSVC parameters in:
23.      // rcx        : root_register_value
24.      // rdx        : new_target
25.      // r8         : function
26.      // r9         : receiver
27.      // [rsp+0x20] : argc
28.      // [rsp+0x28] : argv
29.      __ movq(rdi, arg_reg_3);
30.      __ Move(rdx, arg_reg_2);
31.      // rdi : function
32.      // rdx : new_target
```

```
33.      // Setup the context (we need to use the caller context from the
isolate).
34.      ExternalReference context_address = ExternalReference::Create(
35.          IsolateAddressId::kContextAddress, masm->isolate());
36.      __ movq(rsi, masm->ExternalReferenceAsOperand(context_address));
37.      // Push the function and the receiver onto the stack.
38.      __ Push(rdi);
39.      __ Push(arg_reg_4);
40.      // Current stack contents:
41.      // [rsp + 2 * kSystemPointerSize ... ] : Internal frame
42.      // [rsp + kSystemPointerSize]          : function
43.      // [rsp]                         : receiver
44.      // Current register contents:
45.      // rax : argc
46.      // rbx : argv
47.      // rsi : context
48.      // rdi : function
49.      // rdx : new.target
50.      __ bind(&enough_stack_space);
51.      // Copy arguments to the stack in a loop.
52.      // Invoke the builtin code.
53.      Handle<Code> builtin = is_construct
54.                                 ? BUILTIN_CODE(masm->isolate(), Construct)
55.                                 : masm->isolate()->builtins()->Call();
56.      __ Call(builtin, RelocInfo::CODE_TARGET);
57.    }
58.    __ ret(0);
59.  }
```

上述代码第2行调用`Generate_JSEntryTrampolineHelper(masm, false)`，它的第2个参数后面会用到，该方法的主要作用是把参数压入堆栈。第7-28行代码说明了压入堆栈的参数数量，其中参数function是我们的测试代码。第40-49行说明了堆栈的布局。**提示：** 第40行代码之前有循环压栈的操作。第53行执行masm->isolate()->builtins()->Call()，即Builtin::kCall_ReceiverIsAny。

5 Builtin::kCall_ReceiverIsAny

```
1.  void Builtins::Generate_Call_ReceiverIsAny(MacroAssembler* masm) {
2.    Generate_Call(masm, ConvertReceiverMode::kAny);
3.  }
4.  //...........分隔线.................
5.  void Builtins::Generate_Call(MacroAssembler* masm, ConvertReceiverMode mode) {
6.    // ----------- S t a t e -------------
7.    //  -- rax : the number of arguments (not including the receiver)
8.    //  -- rdi : the target to call (can be any Object)
9.    // -----------------------------------
10.    StackArgumentsAccessor args(rsp, rax);
11.    Label non_callable;
12.    __ JumpIfSmi(rdi, &non_callable);
13.    __ CmpObjectType(rdi, JS_FUNCTION_TYPE, rcx);
14.    __ Jump(masm->isolate()->builtins()->CallFunction(mode),
```

```
15.            RelocInfo::CODE_TARGET, equal);
16.    __ CmpInstanceType(rcx, JS_BOUND_FUNCTION_TYPE);
17.    __ Jump(BUILTIN_CODE(masm->isolate(), CallBoundFunction),
18.            RelocInfo::CODE_TARGET, equal);
19.    // Check if target has a [[Call]] internal method.
20.    __ testb(FieldOperand(rcx, Map::kBitFieldOffset),
21.             Immediate(Map::IsCallableBit::kMask));
22.    __ j(zero, &non_callable, Label::kNear);
23.    // Check if target is a proxy and call CallProxy external builtin
24.    __ CmpInstanceType(rcx, JS_PROXY_TYPE);
25.    __ Jump(BUILTIN_CODE(masm->isolate(), CallProxy), RelocInfo::CODE_TARGET,
26.            equal);
27.  }
```

上述代码第2行调用`Generate_Call(masm, ConvertReceiverMode::kAny)`。第13行代码rdi的值是测试代码的JSFunction，rcx的值是JSFunction的map，`CmpObjectType()`的结果为真，即rdi的类型是`JS_FUNCTION_TYPE`，所以执行第14行代码，进入Builtin::kCallFunction_ReceiverIsAny。

6 Builtin::kCallFunction_ReceiverIsAny

```
1.  void Builtins::Generate_CallFunction_ReceiverIsAny(MacroAssembler* masm) {
2.    Generate_CallFunction(masm, ConvertReceiverMode::kAny);
3.  }
4.  //.............分隔线.................
5.  void Builtins::Generate_CallFunction(MacroAssembler* masm,
6.                                       ConvertReceiverMode mode) {
7.    StackArgumentsAccessor args(rsp, rax);
8.    __ AssertFunction(rdi);
9.    {
10.    // ----------- S t a t e -------------
11.    //  -- rax : the number of arguments (not including the receiver)
12.    //  -- rdx : the shared function info.
13.    //  -- rdi : the function to call (checked to be a JSFunction)
14.    //  -- rsi : the function context.
15.    // ----------------------------------
16.    __ movzxwq(
17.        rbx, FieldOperand(rdx,
SharedFunctionInfo::kFormalParameterCountOffset));
18.    ParameterCount actual(rax);
19.    ParameterCount expected(rbx);
20.    __ InvokeFunctionCode(rdi, no_reg, expected, actual, JUMP_FUNCTION);
21.    // The function is a "classConstructor", need to raise an exception.
22.    __ bind(&class_constructor);
23.    {
24.      FrameScope frame(masm, StackFrame::INTERNAL);
25.      __ Push(rdi);
26.      __ CallRuntime(Runtime::kThrowConstructorNonCallableError);
27.    }
28.  }
```

上述代码第2行调用Generate_CallFunction(masm, ConvertReceiverMode::kAny)。第7-19行代码检测参数并把参数压入堆栈，第10-14行说明了寄存器的值，其中rdi的值是测试代码的JSFuntcion、rdx是对应的SharedFunction。进入第20行代码，源码如下：

7 InvokeFunctionCode

```
1.  void MacroAssembler::InvokeFunctionCode(Register function, Register
new_target,
2.                                          const ParameterCount& expected,
3.                                          const ParameterCount& actual,
4.                                          InvokeFlag flag) {
5.    // On function call, call into the debugger if necessary.
6.    CheckDebugHook(function, new_target, expected, actual);
7.    // Clear the new.target register if not given.
8.    if (!new_target.is_valid()) {
9.      LoadRoot(rdx, RootIndex::kUndefinedValue);
10.    }
11.    Label done;
12.    bool definitely_mismatches = false;
13.    InvokePrologue(expected, actual, &done, &definitely_mismatches, flag,
14.                   Label::kNear);
15.    if (!definitely_mismatches) {
16.      // We call indirectly through the code field in the function to
17.      // allow recompilation to take effect without changing any of the
18.      // call sites.
19.      static_assert(kJavaScriptCallCodeStartRegister == rcx, "ABI mismatch");
20.      LoadTaggedPointerField(rcx,
21.                             FieldOperand(function, JSFunction::kCodeOffset));
22.      if (flag == CALL_FUNCTION) {
23.        CallCodeObject(rcx);
24.      } else {
25.        DCHECK(flag == JUMP_FUNCTION);
26.        JumpCodeObject(rcx);
27.      }
28.      bind(&done);
29.    }
30.  }
```

上述代码第6行是Debug相关的操作，请自行查阅。第20行代码加载JSFunction::kCodeOffset到rcx，第23行代码调用rcx。rcx就是我们念念已久的**Builtins::kInterpreterEntryTrampoline。**
下篇文章讲解InterpreterEntryTrampoline，最后附上官方文档对InterpreterEntryTrampoline的描述:“When the function is called at runtime, the InterpreterEntryTrampoline stub is entered. This stub set up an appropriate stack frame, and then dispatch to the interpreter’s bytecode handler for the function’s first bytecode in order to start execution of the function in the interpreter. The end of each bytecode handler directly dispatches to the next handler via an index into the global interpreter table, based on the bytecode”。

技术总结
** （1）**因为有填充信息，所以stub_entry不等于code入口；

（2）目标代码分为kCallable和kRunMicrotasks两种类型，对应的Ignition流程也略微不同；
（3）大部分的类汇编操作的是由MacroAssembler实现。
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 知乎：https://www.zhihu.com/people/v8blink
本文由灰豆原创发布
转载出处： https://www.anquanke.com/post/id/260982
安全客 - 有思想的安全新媒体