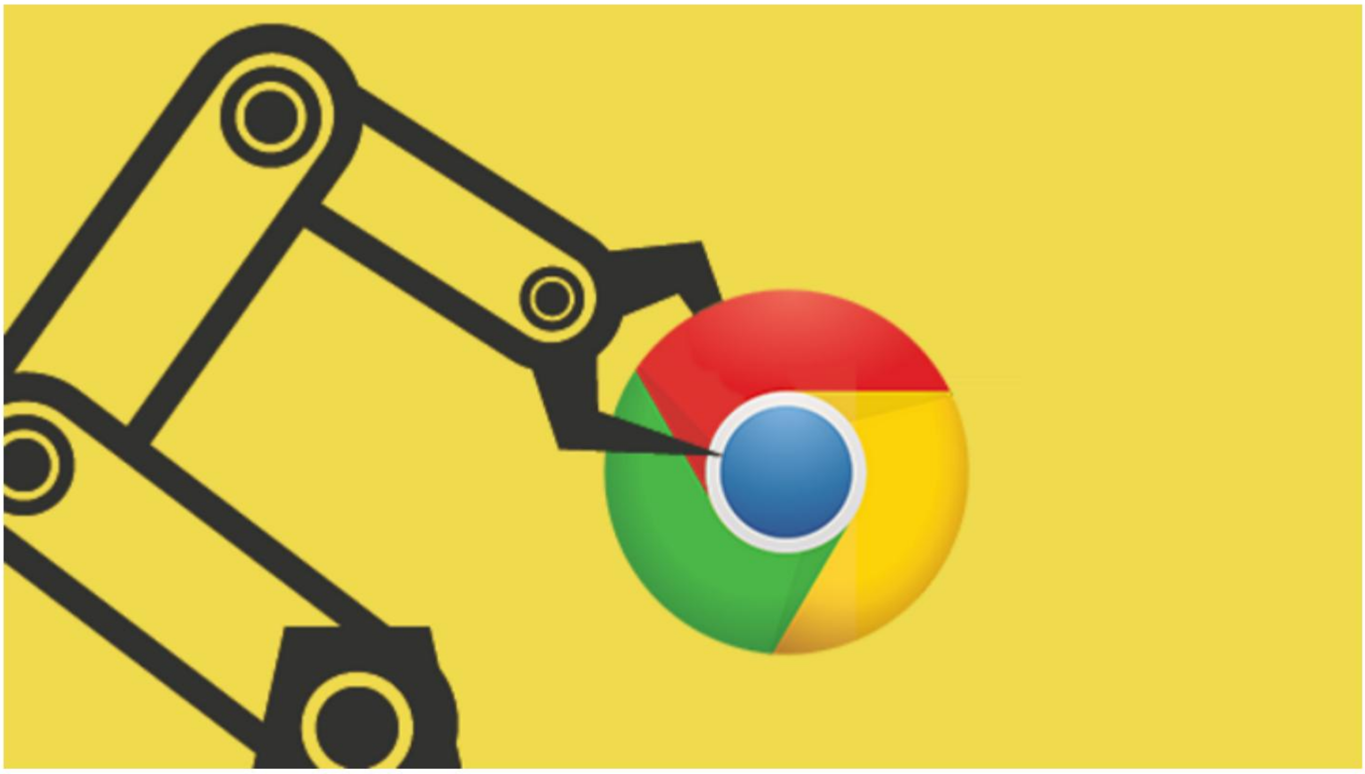
"Chrome V8 Source Code" 30.What exactly does Ignition do?



1 abstract

This article is the fifth of the Builtin topic. It explains the preparation work that needs to be done before Ignition interprets Bytecode. These tasks are completed by a series of Builtins. The work includes: building stacks, pushing parameters, etc. This article analyzes this series of Builtin workflows to understand what preparations Ignition has done for Bytecode.

2 Overview of Ignition

Tip: The V8 version used in this article is 7.9.10, CPU: x64, Builtins-x64.cc. The test sample is as follows:

```
function ignition(s) { this.slogan=s;
        this.start=function()
        {eval('console.log(this.slogan);')}

} worker = new ignition("here we go!"); worker.start();
```

Let's start with V8_WARN_UNUSED_RESULT MaybeHandle<Object> Invoke(Isolate* isolate,const InvokeParams& params), because Ignition will start working from here, the source code is as follows:

```
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. //Omit...........................
12.    return params;
13. }
14. //...........Separator line.......................
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) {
twenty one.   using JSEntryFunction = GeneratedCode<Address(
twenty two.       Address root_register_value, Address new_target, Address target,
twenty three.     Address receiver, intptr_t argc, Address** argv)>;
twenty four.  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.  }
```

The key points of the above code are as follows:

(1) Lines 1-13 of code SetUpForCall() encapsulates parameter params, line 6 of code target is the JSFunction of the test code, lines 8 and 9 The value of the code is 0 and nullptr;

(2) The value of code in line 17 is the entry address of Builtin::kJSEntry;

(3) The value of stub_entry in line 24 is the first instruction position of Builtin::kJSEntry;

(4) The code func in line 27 is the JSFunction of the test sample;

(5) Line 21 of the code stub_entry.Call declares that the third parameter target of stub_entry.Call is func, which is the JSFunction of the test code;

(6) Line 31 of the code stub_entry.Call starts executing Builtin::kJSEntry.

Before entering Builtin::kJSEntry, let's first explain several JSFunction members used by Ignition:

(1) SharedFunction member, which stores the SharedFunction instance corresponding to JSFunciton. The position offset of this member is kSharedFunctionInfoOffset;

(2) code member, which stores Builtins::kInterpreterEntryTrampoline, and the position offset of this member is kCodeOffset;

(3) context member, which stores the context used this time. The position offset of this member is kContextOffset.

Let’s start with Builtin::kJSEntry.

3 Builtin::kJSEntry

```
void Builtins::Generate_JSEntry(MacroAssembler* masm) {
    Generate_JSEntryVariant(masm, StackFrame::ENTRY,
                            Builtins::kJSEntryTrampoline);
}
//............Separator line............
void Generate_JSEntryVariant(MacroAssembler* masm, StackFrame::Type type,
                             Builtins::Name entry_trampoline) {
    Label invoke, handler_entry, exit;
    Label not_outermost_js, not_outermost_js_2;
     { // NOLINT. Scope block confuses linter.
       NoRootArrayScope uninitialized_root_register(masm);
//............Omitted....................
       // Initialize the root register.
       // C calling convention. The first argument is passed in arg_reg_1.
       __ movq(kRootRegister, arg_reg_1);
     // Save copies of the top frame descriptor on the stack.
     ExternalReference c_entry_fp = ExternalReference::Create(
         IsolateAddressId::kCEntryFPAddress, masm->isolate());
     {
       Operand c_entry_fp_operand = masm->ExternalReferenceAsOperand(c_entry_fp);
       __Push (c_entry_fp_operand);
     }
     // Store the context address in the previously-reserved slot.
     ExternalReference context_address = ExternalReference::Create(
         IsolateAddressId::kContextAddress, masm->isolate());
     __ Load(kScratchRegister, context_address);
     static constexpr int kOffsetToContextSlot = -2 * kSystemPointerSize;
     __ movq(Operand(rbp, kOffsetToContextSlot), kScratchRegister);
     __jmp (&invoke);
     __ bind(&handler_entry);
     // Store the current pc as the handler offset. It's used later to create the
     // handler table.
     masm->isolate()->builtins()->SetJSEntryHandlerOffset(handler_entry.pos());
     // Caught exception: Store result (exception) in the pending exception
     // field in the JSEnv and return a failure sentinel.
     ExternalReference pending_exception = ExternalReference::Create(
         IsolateAddressId::kPendingExceptionAddress, masm->isolate());
     __ Store(pending_exception, rax);
     __ LoadRoot(rax, RootIndex::kException);
     __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. //............Omitted......
50. }
```

Line 2 of the above code enters the Generate_JSEntryVariant() method. The third parameter of the method, Builtins::kJSEntryTrampoline, is in line 2. Line 47 will be used. The value of arg_reg_1 in line 15 is isolate->isolate_data()->isolate_root(). Lines 17-22 of code change the current The top frame is pushed onto the stack. Chapters 24-26 load context to ScratchRegister. Lines 36-39 set up exception handling. Lines 46-48 code call Builtins::kJSEntryTrampoline.

4 Builtins::kJSEntryTrampoline

```
1. void Builtins::Generate_JSEntryTrampoline(MacroAssembler* masm) {
2.        Generate_JSEntryTrampolineHelper(masm, false);
3. }
4. //.............Separator line.............
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
twenty one.             // new.target.
twenty two.             // MSVC parameters in:
twenty three.             // rcx //                : root_register_value
twenty four.             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. }
```

The second line of the above code calls Generate_JSEntryTrampolineHelper(masm, false), and its second parameter will be used later. The main function of the method is to push parameters onto the stack. Lines 7-28 of the code illustrate the number of parameters pushed onto the stack, where the parameter function is our test Try the code. Lines 40-49 illustrate the layout of the stack. Tip: There is a loop push operation before line 40 of code. Line 53 executes masm->isolate()->builtins()->Call(), namely Builtin::kCall_ReceiverIsAny.

5 Builtin::kCall_ReceiverIsAny

```
void Builtins::Generate_Call_ReceiverIsAny(MacroAssembler* masm) {
    Generate_Call(masm, ConvertReceiverMode::kAny);
}
//.............Separator line.............
void Builtins::Generate_Call(MacroAssembler* masm, ConvertReceiverMode mode) {
    // ----------- S tate -------------
    // -- rax : the number of arguments (not including the receiver)
    // -- rdi : the target to call (can be any Object)
    //-----------------------------------
      StackArgumentsAccessor args(rsp, rax);
      Label non_callable;
      __ JumpIfSmi(rdi, &non_callable);
      __ CmpObjectType(rdi, JS_FUNCTION_TYPE, rcx);
      __ 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),
twenty one.                 Immediate(Map::IsCallableBit::kMask));
twenty two.         __ j(zero, &non_callable, Label::kNear);
twenty three.         // Check if target is a proxy and call CallProxy external builtin
twenty four.         __CmpInstanceType (rcx, JS_PROXY_TYPE);
25.         __ Jump(BUILTIN_CODE(masm->isolate(), CallProxy), RelocInfo::CODE_TARGET,
26.                    equal);
27. }
```

Line 2 of the above code calls Generate_Call(masm, ConvertReceiverMode::kAny). The value of rdi in line 13 is the test code JSFunction, the value of rcx is the map of JSFunction, the result of CmpObjectType() is true, that is, the type of rdi is JS_FUNCTION_TYPE, so execute line 14 of the code and enter Builtin::kCallFunction_ReceiverIsAny.

6 Builtin::kCallFunction_ReceiverIsAny

```
void Builtins::Generate_CallFunction_ReceiverIsAny(MacroAssembler* masm) {
        Generate_CallFunction(masm, ConvertReceiverMode::kAny);
}
//............Separator line............
void Builtins::Generate_CallFunction(MacroAssembler* masm,
                                                   ConvertReceiverMode mode) {
        StackArgumentsAccessor args(rsp, rax);
        __AssertFunction (rdi);
        {
          // ----------- S tate -------------
          // -- rax : the number of arguments (not including the receiver)
          // -- rdx : the shared function info.
          // -- rdi : the function to call (checked to be a JSFunction)
          // -- rsi : the function context.
          //-----------------------------------
          __ movzxwq(
              rbx, FieldOperand(rdx,
SharedFunctionInfo::kFormalParameterCountOffset));
          ParameterCount actual(rax);
          ParameterCount expected(rbx);
          __InvokeFunctionCode (rdi, no_reg, expected, actual, JUMP_FUNCTION);
          // The function is a "classConstructor", need to raise an exception.
          __ bind(&class_constructor);
          {
              FrameScope frame(masm, StackFrame::INTERNAL);
              __Push (rdi);
              __CallRuntime (Runtime::kThrowConstructorNonCallableError);
          }
}
```

Line 2 of the above code calls Generate_CallFunction(masm, ConvertReceiverMode::kAny). Lines 7-19 code detection parameters

Count and push the parameters onto the stack. Lines 10-14 illustrate the register value, where the value of rdi is the corresponding JSFuntcion and rdx of the test code.

SharedFunction. Enter the 20th line of code. The source code is as follows:

7InvokeFunctionCode

```
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,
twenty one.                      FieldOperand(function, JSFunction::kCodeOffset));
twenty two.  if (flag == CALL_FUNCTION) {
twenty three.  CallCodeObject(rcx);
twenty four. } else {
25.         DCHECK(flag == JUMP_FUNCTION);
26.         JumpCodeObject(rcx);
27.       }
28.       bind(&done);
29.     }
30. }
```

Line 6 of the above code is Debug related operations, please check it yourself. Line 20 of code loads JSFunction::kCodeOffset to rcx, line 23

line of code calls rcx. rcx is Builtins::kInterpreterEntryTrampoline that we have been thinking about for a long time.

The next article explains InterpreterEntryTrampoline, and finally attaches the official document's description of InterpreterEntryTrampoline: "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".

Technical summary

(1) Because there is filling information, stub_entry is not equal to the code entry;

(2) The target code is divided into two types: kCallable and kRunMicrotasks, and the corresponding Ignition processes are also slightly

different; **(3)**Most of the class assembly operations are implemented by MacroAssembler.

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

Personal abilities are limited and there are shortcomings and

mistakes. Welcome to criticize and correct me. WeChat: qq9123013 Note: v8 Communication Zhihu: https://

www.zhihu.com/people/

v8blink This article was originally published by Gray Bean and reprinted from: https://

www. anquanke.com/post/id/260982 Anquanke - Thoughtful new security media