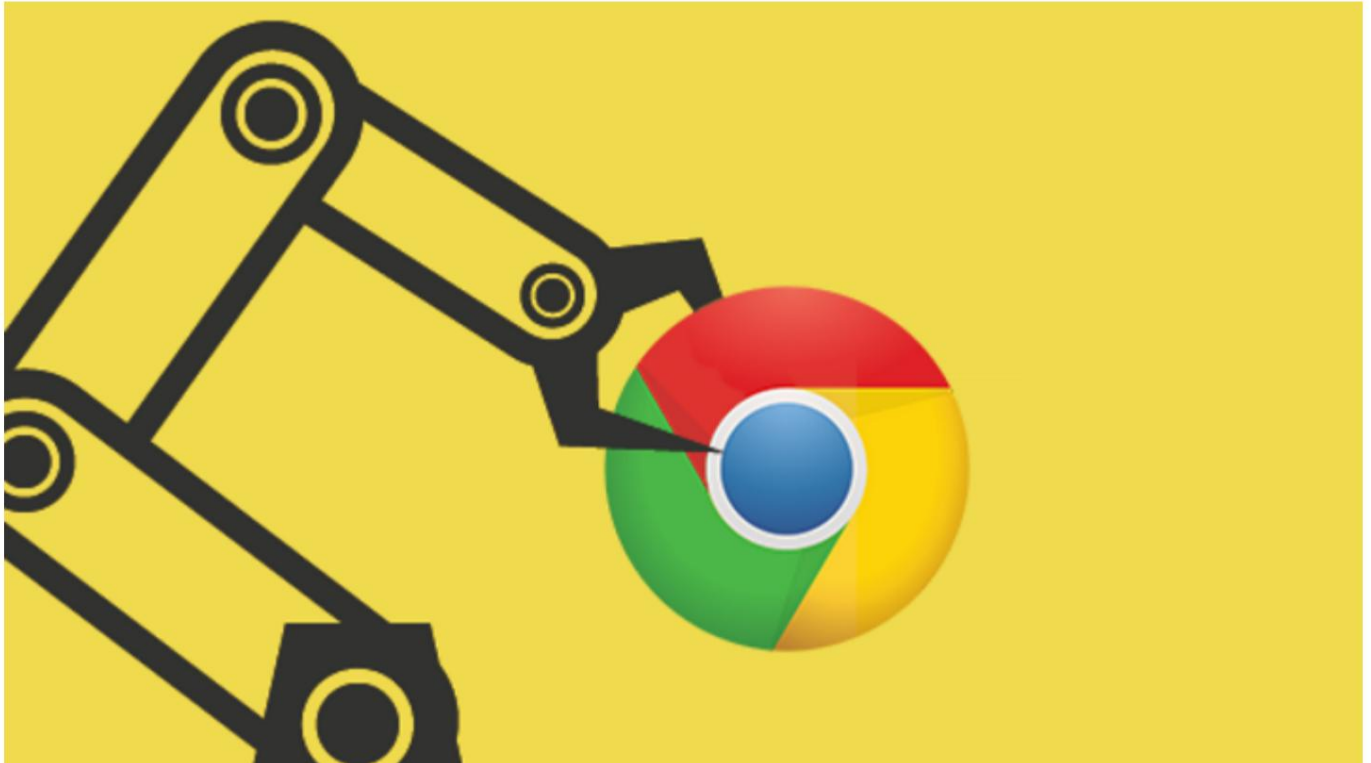


"Chrome V8 Source Code" 32. Bytecode and Compiler Pipeline

detail



1 abstract

This article is the seventh in the Builtin topic. The previous article explained the `Builtin::kInterpreterEntryTrampoline` source code. This article will introduce it. Builtin's compilation process, during which you can see the technical details of the code generated by Bytecode handler, and you can also use this process to understand Compiler Pipeline technology and important data structures.

2 Important data structures of Bytecode handler

`GenerateBytecodeHandler()` is responsible for generating Bytecode handler. The source code is as follows:

```
1. Handle<Code> GenerateBytecodeHandler(Isolate* isolate, const char* debug_name,
2.                                     Bytecode bytecode,
3.                                     OperandScale operand_scale,
4.                                     int builtin_index,
5.                                     const AssemblerOptions& options) {
6.   Zone zone(isolate->allocator(), ZONE_NAME);
7.   compiler::CodeAssemblerState state(
8.     isolate, &zone, InterpreterDispatchDescriptor{}, Code::BYTECODE_HANDLER,
9.     debug_name,
10.    FLAG_untrusted_code_mitigations
11.    ? PoisoningMitigationLevel::kPoisonCriticalOnly
12.    : PoisoningMitigationLevel::kDontPoison,
```

```

13.         builtin_index);
14.     switch (bytecode) {
15. #define CALL_GENERATOR(Name, ...) case                                \
        Bytecode::k##Name: 16.                                         \
17.         Name##Assembler::Generate(&state, operand_scale); \
18.         break;
19.         BYTECODE_LIST(CALL_GENERATOR);
20. #undef CALL_GENERATOR
twenty one: }
twenty two:     Handle<Code> code = compiler::CodeAssembler::GenerateCode(&state, options);
23. #ifdef ENABLE_DISASSEMBLER
twenty four:     if (FLAG_trace_ignition_codegen) {
25.         StdoutStream os;
26.         code->Disassemble(Bytecodes::ToString(bytecode), os);
27.         os << std::flush;
28.     }
29. #endif // ENABLE_DISASSEMBLER
30.     return code;
31. }

```

Lines 7-13 of the above code initialize the state. The state includes BytecodeOffset, DispatchTable and Descriptor. Bytecode compiles state will be used when. Lines 14-21 generate the Bytecode handler source code. Line 17 state is passed into GenerateCode() as a parameter, used to record the generation results of Bytecode handler. The following uses LdaSmi as an example to explain the important data structure of Bytecode handler:

```

IGNITION_HANDLER(LdaSmi, InterpreterAssembler) {
    TNode<Smi> smi_int = BytecodeOperandImmSmi(0);
    SetAccumulator(smi_int);
    Dispatch();
}

```

The above code sets the value of the accumulation register to smi. After expanding the macro IGNITION_HANDLER, you can see that LdaSmiAssembler is a subclass. InterpreterAssembler is the parent class, described as follows:

(1) LdaSmiAssembler includes the entry method Genrate() to generate LdaSmi. The source code is as follows:

```

1. void Name##Assembler::Generate(compiler::CodeAssemblerState* state,
1.         OperandScale scale) {
2. Name##Assembler assembler(state, Bytecode::k##Name, scale);
3. state->SetInitialDebugInformation(#Name, __FILE__, __LINE__);
4. assembler.GenerateImpl(); 6.}

```

The third line of code above creates an LdaSmiAssembler instance. The fourth line of code writes debug information into state.

(2) InterpreterAssembler provides interpreter-related functions. The source code is as follows:

```

1. class V8_EXPORT_PRIVATE InterpreterAssembler : public CodeStubAssembler {
2. public:

```

```
3. //.....omitted.....
4.private :
5. TNode<BytecodeArray> BytecodeArrayTaggedPointer();
6. TNode<ExternalReference> DispatchTablePointer();
7. TNode<Object> GetAccumulatorUnchecked();
8. TNode<RawPtrT> GetInterpretedFramePointer();
9.     compiler::TNode<IntPtrT> RegisterLocation(Register reg);
10.     compiler::TNode<IntPtrT> RegisterLocation(compiler::TNode<IntPtrT>
reg_index);
11. compiler::TNode<IntPtrT> NextRegister(compiler::TNode<IntPtrT> reg_index);
12.     compiler::TNode<Object> LoadRegister(compiler::TNode<IntPtrT> reg_index);
13.     void StoreRegister(compiler::TNode<Object> value,
14.                         compiler::TNode<IntPtrT> reg_index);
15.     void CallPrologue();
16.     void CallEpilogue();
17.     void TraceBytecodeDispatch(TNode<WordT> target_bytecode);
18.     void TraceBytecode(Runtime::FunctionId function_id);
19.     void Jump(compiler::TNode<IntPtrT> jump_offset, bool backward);
20.     void JumpConditional(compiler::TNode<BoolT> condition,
                           compiler::TNode<IntPtrT> jump_offset);
twenty one.     void SaveBytecodeOffset();
twenty two.     TNode<IntPtrT> ReloadBytecodeOffset();
twenty three.     TNode<IntPtrT> Advance();
twenty four.     TNode<IntPtrT> Advance(int delta);
25.     TNode<IntPtrT> Advance(TNode<IntPtrT> delta, bool backward = false);
26.     compiler::TNode<WordT> LoadBytecode(compiler::TNode<IntPtrT>
bytecode_offset);
27.     void DispatchToBytecodeHandlerEntry(compiler::TNode<RawPtrT> handler_entry,
28.                                           compiler::TNode<IntPtrT>
bytecode_offset);
29.     int CurrentBytecodeSize() const;
30.     OperandScale operand_scale() const { return operand_scale_; }
31.     Bytecode bytecode_;
32.     OperandScale operand_scale_;
33.     CodeStubAssembler::TVariable<RawPtrT> interpreted_frame_pointer_;
34.     CodeStubAssembler::TVariable<BytecodeArray> bytecode_array_;
35.     CodeStubAssembler::TVariable<IntPtrT> bytecode_offset_;
36.     CodeStubAssembler::TVariable<ExternalReference> dispatch_table_;
37.     CodeStubAssembler::TVariable<Object> accumulator_;
38.     AccumulatorUse accumulator_use_;
39.     bool made_call_;
40.     bool reloaded_frame_ptr_;
41.     bool bytecode_array_valid_;
42.     DISALLOW_COPY_AND_ASSIGN(InterpreterAssembler);
43. };
44. };
```

The 5th line of code above obtains the address of BytecodeArray; the 6th line of code obtains the address of DispatchTable; the 7th line of code obtains the cumulative dispatch

The value of the register; lines 8-13 of code are used to operate the register; lines of code 15-16 are used for stack processing before and after calling the function; lines of code 17-18 are used

For tracking Bytecode, line 18 calls Runtime::Runtime_InterpreterTraceBytecodeEntry to output register information; line 18

Lines 19-20 of code are two jump instructions. Advance (line 24-26) is called inside the instruction to complete the jump operation; lines 24-26 of code

Used to obtain the next Bytecode; the member variables defined in lines 32-42 of the code will be frequently used in the Bytecode handler, for example, in

In SetAccumulator(zero_value), first set accumulator_use_ to write status, and then write the value to accumulator_.

(3) CodeStubAssembler is the parent class of InterpreterAssembler and provides JavaScript-specific methods. The source code is as follows:

```
1. class V8_EXPORT_PRIVATE CodeStubAssembler: public compiler::CodeAssembler,
2.     public TorqueGeneratedExportedMacrosAssembler {
3. public:
4. TNode<Int32T> StringCharCodeAt(SloppyTNode<String> string,
5.                               SloppyTNode<IntPtrT> index);
6. TNode<String> StringFromSingleCharCode(TNode<Int32T> code);
7. TNode<String> SubString(TNode<String> string, TNode<IntPtrT> from,
8.                        TNode<IntPtrT> to);
9. TNode<String> StringAdd(Node* context, TNode<String> first,
10.                       TNode<String> second);
11. TNode<Number> ToNumber(
12.     SloppyTNode<Context> context, SloppyTNode<Object> input,
13.     BigIntHandling bigint_handling = BigIntHandling::kThrow);
14. TNode<Number> ToNumber_Inline(SloppyTNode<Context> context,
15.                              SloppyTNode<Object> input);
16. TNode<BigInt> ToBigInt(SloppyTNode<Context> context,
17.                       SloppyTNode<Object> input);
18. TNode<Number> ToUint32(SloppyTNode<Context> context,
19.                      SloppyTNode<Object> input);
20. // ES6 7.1.17 ToIndex, but jumps to range_error if the result is not a Smi.
    TNode<Smi> ToSmiIndex(TNode<Context> context, TNode<Object> input,
    Label* range_error);
    TNode<Smi> ToSmiLength(TNode<Context> context, TNode<Object> input,
    Label* range_error);
25. TNode<Number> ToLength_Inline(SloppyTNode<Context> context,
26.                              SloppyTNode<Object> input);
27. TNode<Object> GetProperty(SloppyTNode<Context> context,
28.                          SloppyTNode<Object> receiver, Handle<Name> name)
{}
29. TNode<Object> GetProperty(SloppyTNode<Context> context,
30.                          SloppyTNode<Object> receiver,
31.                          SloppyTNode<Object> name) {}
32. TNode<Object> SetPropertyStrict(TNode<Context> context,
33.                                TNode<Object> receiver, TNode<Object> key,
34.                                TNode<Object> value) {}
35. template <class... TArgs>
36. TNode<Object> CallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
37.                          TArgs... args) {}
38. template <class... TArgs>
39. void TailCallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
40.                     TArgs... args) {}
41. void LoadPropertyFromFastObject(...Omit parameters...);
42. void LoadPropertyFromFastObject(...Omit parameters...);
43. void LoadPropertyFromNameDictionary(...Omit parameters...);
44. void LoadPropertyFromGlobalDictionary(...Omit parameters...);
45. void UpdateFeedback(Node* feedback, Node* feedback_vector, Node* slot_id);
46. void ReportFeedbackUpdate(TNode<FeedbackVector> feedback_vector,
47.                           SloppyTNode<IntPtrT> slot_id, const char*
reason);
```

```
48. void CombineFeedback(Variable* existing_feedback, int feedback);
49. void CombineFeedback(Variable* existing_feedback, Node* feedback);
50. void OverwriteFeedback(Variable* existing_feedback, int new_feedback);
51. void BranchIfNumberRelationalComparison(Operation op,
52.                                     SloppyTNode<Number> left,
53.                                     SloppyTNode<Number> right,
54.                                     Label* if_true, Label* if_false);
55. void BranchIfNumberEqual(TNode<Number> left, TNode<Number> right,
56.                           Label* if_true, Label* if_false) {
57. }
58. };
```

CodeStubAssembler uses assembly language to implement JavaScript's unique methods. The base class CodeAssembler encapsulates assembly language.

CodeStubAssembler uses the assembly functions provided by CodeAssembler to implement string conversion, attribute acquisition, branch jumping, etc.

JavaScript functionality, that's what CodeStubAssembler is all about.

Lines 4-9 of the above code implement string related operations; lines 11-18 of code implement type conversion; lines 21-26 implement the ES specification

The function; Lines 27-38 implement getting and setting properties; Lines 39-43 implement the calling methods of Builtin and Runtime API; Lines 45-50

Lines of code are used to manage Feedback; Lines 51-55 implement the IF function. **(4)** CodeAssembler encapsulates the assembly function and implements

Branch, Goto and other functions, the source code is as follows:

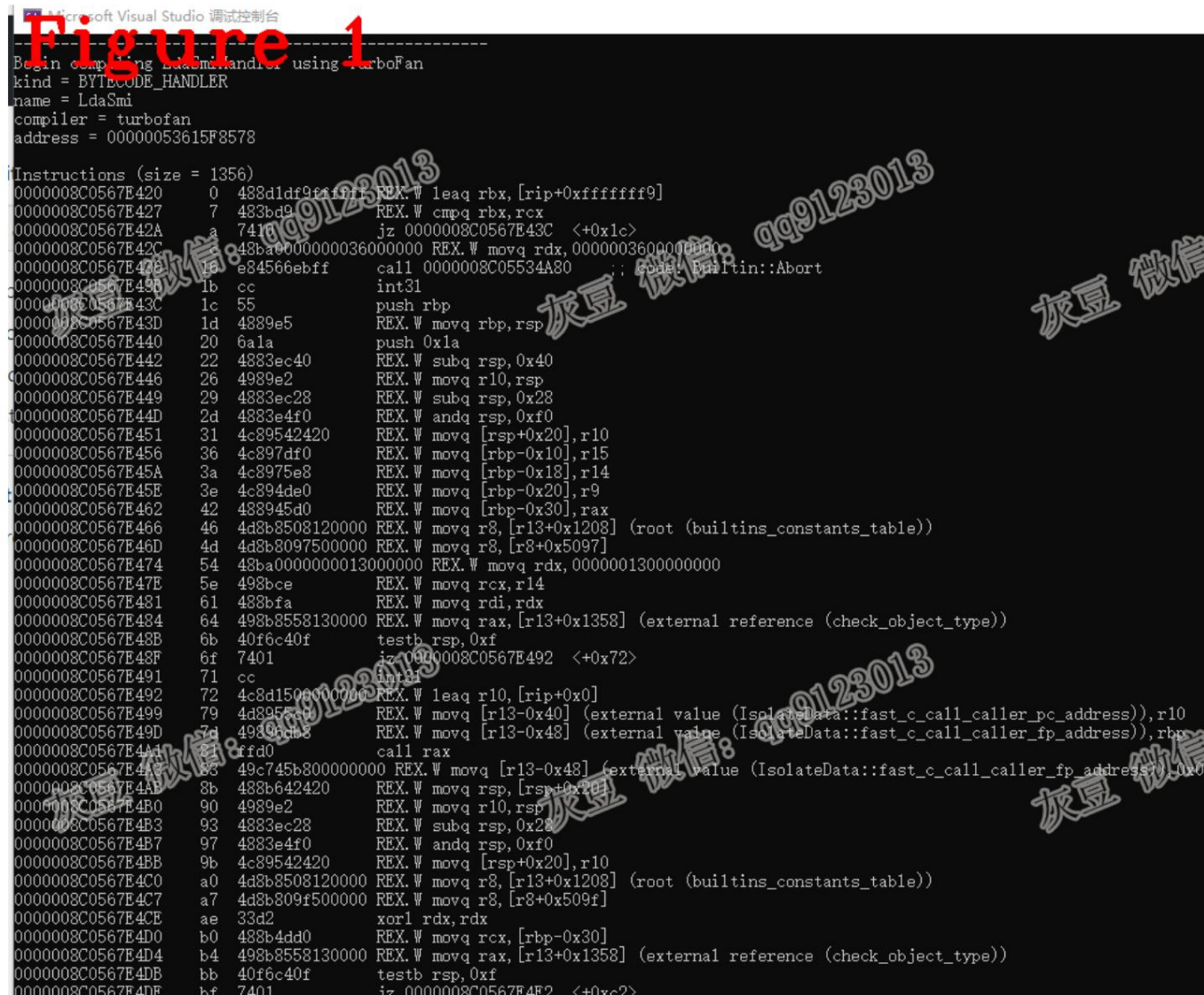
```
1. class V8_EXPORT_PRIVATE CodeAssembler {
2.     void Branch(TNode<BoolT> condition,
3.                 CodeAssemblerParameterizedLabel<T...>* if_true,
4.                 CodeAssemblerParameterizedLabel<T...>* if_false, Args... args) {
5.         if_true->AddInputs(args...);
6.         if_false->AddInputs(args...);
7.         Branch(condition, if_true->plain_label(), if_false->plain_label());
8.     }
9.     template <class... T, class... Args>
10.    void Goto(CodeAssemblerParameterizedLabel<T...>* label, Args... args) {
11.        label->AddInputs(args...);
12.        Goto(label->plain_label());
13.    }
14.    void Branch(TNode<BoolT> condition, const std::function<void()>& true_body,
15.                const std::function<void()>& false_body);
16.    void Branch(TNode<BoolT> condition, Label* true_label,
17.                const std::function<void()>& false_body);
18.    void Branch(TNode<BoolT> condition, const std::function<void()>& true_body,
19.                Label* false_label);
20.    void Switch(Node* index, Label* default_label, const int32_t* case_values,
                Label** case_labels, size_t case_count);
    twenty one.
    twenty two. }
```

3 Compiler Pipeline

Line 22 of GenerateBytecodeHandler() completes the compilation of Bytecode LdaSmi. The source code is as follows:

```
1. Handle<Code> CodeAssembler::GenerateCode(CodeAssemblerState* state,
2.                                     const AssemblerOptions& options) {
3. RawMachineAssembler* rasm = state->raw_assembler_.get();
4. Handle<Code> code;
5. Graph* graph = rasm->ExportForOptimization();
6. code = Pipeline::GenerateCodeForCodeStub(...Omit parameters...)
7.     .ToHandleChecked();
8. state->code_generated_ = true;
9. return code;
10. }
11. //.....Separator line.....
12. MaybeHandle<Code> Pipeline::GenerateCodeForCodeStub(...Omit parameters...) {
13.     OptimizedCompilationInfo info(CStrVector(debug_name), graph->zone(), kind);
14.     info.set_builtin_index(builtin_index);
15.     if (poisoning_level != PoisoningMitigationLevel::kDontPoison) {
16.         info.SetPoisoningMitigationLevel(poisoning_level);
17.     }
18.     // Construct a pipeline for scheduling and code generation.
19.     ZoneStats zone_stats(isolate->allocator());
20.     NodeOriginTable node_origins(graph);
    twenty one.     JumpOptimizationInfo jump_opt;
    twenty two.     bool should_optimize_jumps =
    twenty three.         isolate->serializer_enabled() && FLAG_turbo_rewrite_far_jumps;
    twenty four.     PipelineData data(&zone_stats, &info, isolate, isolate->allocator(), graph,
25.                                nullptr, source_positions, &node_origins,
26.                                should_optimize_jumps ? &jump_opt : nullptr, options);
27.     data.set_verify_graph(FLAG_verify_csa);
28.     std::unique_ptr<PipelineStatistics> pipeline_statistics;
29.     if (FLAG_turbo_stats || FLAG_turbo_stats_nvp) {
30.     }
31.     PipelineImpl pipeline(&data);
32.     if (info.trace_turbo_json_enabled() || info.trace_turbo_graph_enabled())
    {///..Omit...
33.     }
34.     pipeline.Run<CsaEarlyOptimizationPhase>();
35.     pipeline.RunPrintAndVerify(CsaEarlyOptimizationPhase::phase_name(), true);
36.     // .....omitted.....
37.     PipelineData second_data(...parameters omitted...);
38.     second_data.set_verify_graph(FLAG_verify_csa);
39.     PipelineImpl second_pipeline(&second_data);
40.     second_pipeline.SelectInstructionsAndAssemble(call_descriptor);
41.     Handle<Code> code;
42.     if (jump_opt.is_optimizable()) {
43.         jump_opt.set_optimizing();
44.         code = pipeline.GenerateCode(call_descriptor).ToHandleChecked();
45.     } else {
46.         code = second_pipeline.FinalizeCode().ToHandleChecked();
47.     }
48.     return code;
49. }
```


The above 6th line of code enters the Pipeline to start the compilation work; 13-29 is used to set Pipeline information; the enable tag on line 32 is defined in flag-definitions.h, they use Json to output the current compilation information; 34-40 This line of code implements functions such as generating the initial assembly code, optimizing the initial assembly code, and using the optimized data to regenerate the final code. Note that the 36th line of code omits the optimization of the initial assembly code. Figure 1 shows the compilation results of LdaSmi.



```
Microsoft Visual Studio 调试控制台
-----
Begin compiling LdaSmiHandler using TurboFan
kind = BYTECODE_HANDLER
name = LdaSmi
compiler = turbofan
address = 00000053615F8578

Instructions (size = 1356)
00000008C0567E420 0 488d1df9f1f1f1f1 REX.W leaq rbx, [rip+0xffffffff9]
00000008C0567E427 7 483bd9 REX.W cmpq rbx, rcx
00000008C0567E42A a 7416 jz 00000008C0567E43C <+0x1c>
00000008C0567E42C 2 48ba9000000036000000 REX.W movq rdx, 0000003600000000
00000008C0567E42E 1a e84566ebff call 00000008C05534A80 ; Code: Builtin::Abort
00000008C0567E430 1b cc int3!
00000008C0567E43C 1c 55 push rbp
00000008C0567E43D 1d 4839e5 REX.W movq rbp, rsp
00000008C0567E440 20 6a1a push 0x1a
00000008C0567E442 22 483ec40 REX.W subq rsp, 0x40
00000008C0567E446 26 4939e2 REX.W movq r10, rsp
00000008C0567E449 29 483ec28 REX.W subq rsp, 0x28
00000008C0567E44D 2d 483e4f0 REX.W andq rsp, 0xf0
00000008C0567E451 31 4c89542420 REX.W movq [rsp+0x20], r10
00000008C0567E456 36 4c897df0 REX.W movq [rbp-0x10], r15
00000008C0567E45A 3a 4c8975e8 REX.W movq [rbp-0x18], r14
00000008C0567E45E 3e 4c894de0 REX.W movq [rbp-0x20], r9
00000008C0567E462 42 483945d0 REX.W movq [rbp-0x30], rax
00000008C0567E466 46 4d3b8508120000 REX.W movq r8, [r13+0x1208] (root (builtins_constants_table))
00000008C0567E46D 4d 4d3b8097500000 REX.W movq r8, [r8+0x5097]
00000008C0567E474 54 48ba0000000013000000 REX.W movq rdx, 0000001300000000
00000008C0567E47E 5e 493bce REX.W movq rcx, r14
00000008C0567E481 61 483bfa REX.W movq rdi, rdx
00000008C0567E484 64 493b8558130000 REX.W movq rax, [r13+0x1358] (external reference (check_object_type))
00000008C0567E48B 6b 40f6c40f testb rsp, 0xf
00000008C0567E48F 6f 7401 iz 00000008C0567E492 <+0x72>
00000008C0567E491 71 cc int3!
00000008C0567E492 72 4c8d1500000000 REX.W leaq r10, [rip+0x0]
00000008C0567E499 79 4d395f50 REX.W movq [r13-0x40] (external value (IsolateData::fast_c_call_caller_pc_address)), r10
00000008C0567E49D 7d 49395f50 REX.W movq [r13-0x48] (external value (IsolateData::fast_c_call_caller_fp_address)), rbp
00000008C0567E4A1 81 ffd0 call rax
00000008C0567E4A3 83 49c745b800000000 REX.W movq [r13-0x48] (external value (IsolateData::fast_c_call_caller_fp_address)), r10
00000008C0567E4AB 8b 483b642420 REX.W movq rsp, [rsp+0x20]
00000008C0567E4B0 90 4939e2 REX.W movq r10, rsp
00000008C0567E4B3 93 483ec28 REX.W subq rsp, 0x28
00000008C0567E4B7 97 483e4f0 REX.W andq rsp, 0xf0
00000008C0567E4BB 9b 4c89542420 REX.W movq [rsp+0x20], r10
00000008C0567E4C0 a0 4d3b8508120000 REX.W movq r8, [r13+0x1208] (root (builtins_constants_table))
00000008C0567E4C7 a7 4d3b8097500000 REX.W movq r8, [r8+0x5097]
00000008C0567E4CE ae 33d2 xorl rdx, rdx
00000008C0567E4D0 b0 483b4dd0 REX.W movq rcx, [rbp-0x30]
00000008C0567E4D4 b4 493b8558130000 REX.W movq rax, [r13+0x1358] (external reference (check_object_type))
00000008C0567E4DB bb 40f6c40f testb rsp, 0xf
00000008C0567E4DF bf 7401 iz 00000008C0567E4E2 <+0xc2>
```

Technical summary

(1) The compilation process of Bytecode Handler can be debugged in V8 only when v8_use_snapshot = false; (2) CodeAssembler encapsulates assembly, CodeStubAssembler encapsulates JavaScript-specific functions, and InterpreterAssembler encapsulates the functions required by the interpreter. Among these three layers of encapsulation, Above is the Bytecode Handler; (3) V8 compiles all Builtins including the Bytecode handler during initialization. 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

This article was originally

published by Huidou with a reprint statement, indicating the source: <https://www.anquanke.com/post/id/262468>

Anquanke - Thoughtful new security media