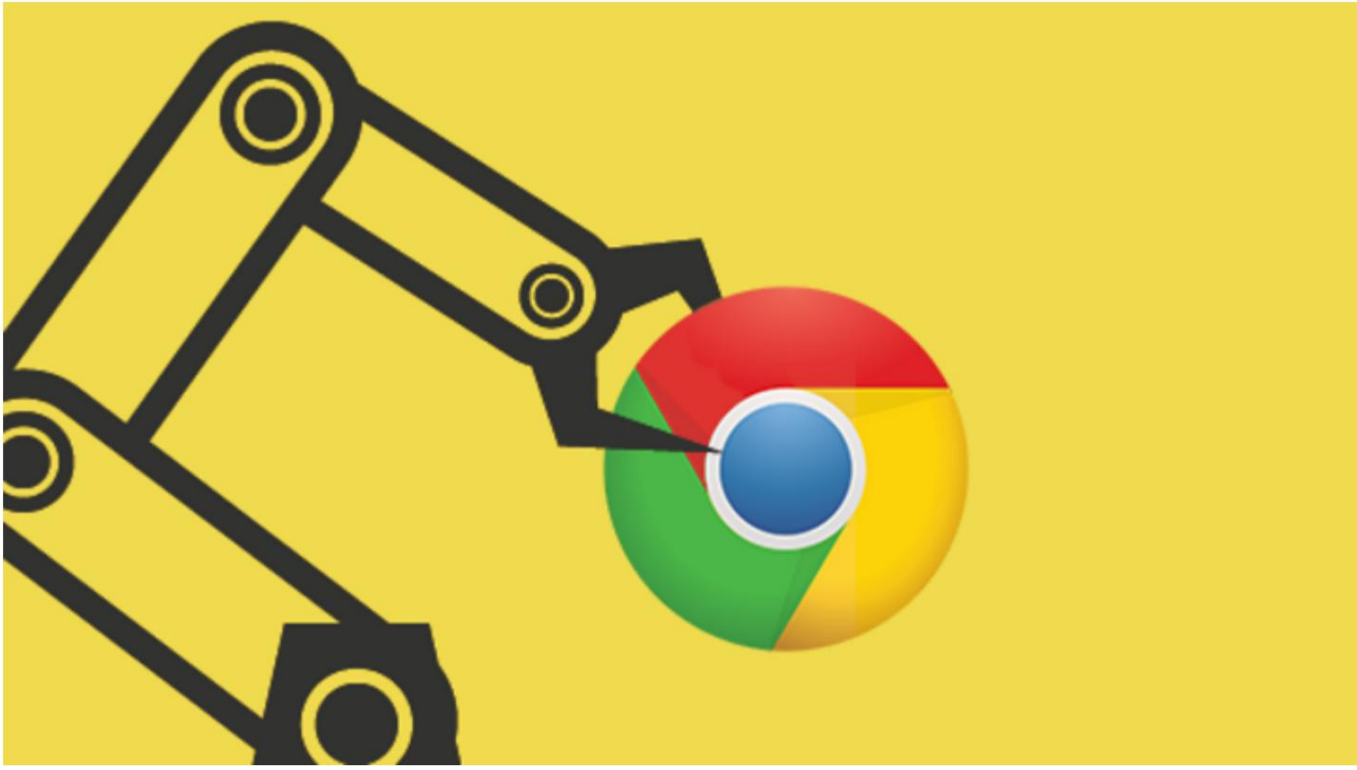
"Chrome V8 Source Code" 32. Bytecode and Compiler Pipeline detail



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. Builin's compilation process, during which you can see the technical details of the code generated by Bytecode hanlder, 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:

```
Handle<Code> GenerateBytecodeHandler(Isolate* isolate, const char* debug_name,
                                     Bytecode bytecode,
                                     OperandScale operand_scale,
                                     int builtin_index,
                                     const AssemblerOptions& options) {
    Zone zone(isolate->allocator(), ZONE_NAME);
    compiler::CodeAssemblerState state(
        isolate, &zone, InterpreterDispatchDescriptor{}, Code::BYTECODE_HANDLER,
        debug_name,
         FLAG_untrusted_code_mitigations
             ? PoisoningMitigationLevel::kPoisonCriticalOnly
             : 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 hadler. 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:

```
class V8_EXPORT_PRIVATE InterpreterAssembler : public CodeStubAssembler {
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,
twenty one.                                compiler::TNode<IntPtrT> jump_offset);
twenty two.      void SaveBytecodeOffset();
twenty three.      TNode<IntPtrT> ReloadBytecodeOffset();
twenty four.      TNode<IntPtrT> Advance();
25.      TNode<IntPtrT> Advance(int delta);
26.      TNode<IntPtrT> Advance(TNode<IntPtrT> delta, bool backward = false);
27.      compiler::TNode<WordT> LoadBytecode(compiler::TNode<IntPtrT>
bytecode_offset);
28.      void DispatchToBytecodeHandlerEntry(compiler::TNode<RawPtrT> handler_entry,
29.                                                  compiler::TNode<IntPtrT>
bytecode_offset);
30.      int CurrentBytecodeSize() const;
31.      OperandScale operand_scale() const { return operand_scale_; }
32.      Bytecode bytecode_;
33.      OperandScale operand_scale_;
34.      CodeStubAssembler::TVariable<RawPtrT> interpreted_frame_pointer_;
35.      CodeStubAssembler::TVariable<BytecodeArray> bytecode_array_;
36.      CodeStubAssembler::TVariable<IntPtrT> bytecode_offset_;
37.      CodeStubAssembler::TVariable<ExternalReference> dispatch_table_;
38.      CodeStubAssembler::TVariable<Object> accumulator_;
39.      AccumulatorUse accumulator_use_;
40.      bool made_call_;
41.      bool reloaded_frame_ptr_;
42.      bool bytecode_array_valid_;
43.      DISALLOW_COPY_AND_ASSIGN(InterpreterAssembler);
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:

```
class V8_EXPORT_PRIVATE CodeStubAssembler: public compiler::CodeAssembler,
              public TorqueGeneratedExportedMacrosAssembler {
public:
TNode<Int32T> StringCharCodeAt(SloppyTNode<String> string,
                                 SloppyTNode<IntPtrT> index);
    TNode<String> StringFromSingleCharCode(TNode<Int32T> code);
    TNode<String> SubString(TNode<String> string, TNode<IntPtrT> from,
                            TNode<IntPtrT> to);
    TNode<String> StringAdd(Node* context, TNode<String> first,
                            TNode<String> second);
      TNode<Number> ToNumber(
          SloppyTNode<Context> context, SloppyTNode<Object> input,
          BigIntHandling bigint_handling = BigIntHandling::kThrow);
      TNode<Number> ToNumber_Inline(SloppyTNode<Context> context,
                                    SloppyTNode<Object> input);
      TNode<BigInt> ToBigInt(SloppyTNode<Context> context,
                             SloppyTNode<Object> input);
      TNode<Number> ToUint32(SloppyTNode<Context> context,
                             SloppyTNode<Object> input);
      // 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);
      TNode<Number> ToLength_Inline(SloppyTNode<Context> context,
                                    SloppyTNode<Object> input);
      TNode<Object> GetProperty(SloppyTNode<Context> context,
                                SloppyTNode<Object> receiver, Handle<Name> name)
{}
      TNode<Object> GetProperty(SloppyTNode<Context> context,
                                SloppyTNode<Object> receiver,
                                SloppyTNode<Object> name) {}
      TNode<Object> SetPropertyStrict(TNode<Context> context,
                                      TNode<Object> receiver, TNode<Object> key,
                                      TNode<Object> value) {}
      template <class... TArgs>
      TNode<Object> CallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
                                TArgs... args) {}
      template <class... TArgs>
      void TailCallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
                           TArgs... args) { }
      void LoadPropertyFromFastObject(...Omit parameters...);
      void LoadPropertyFromFastObject(...Omit parameters...);
      void LoadPropertyFromNameDictionary(...Omit parameters...);
      void LoadPropertyFromGlobalDictionary(...Omit parameters...);
      void UpdateFeedback(Node* feedback, Node* feedback_vector, Node* slot_id);
      void ReportFeedbackUpdate(TNode<FeedbackVector> feedback_vector,
                                SloppyTNode<UintPtrT> 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:

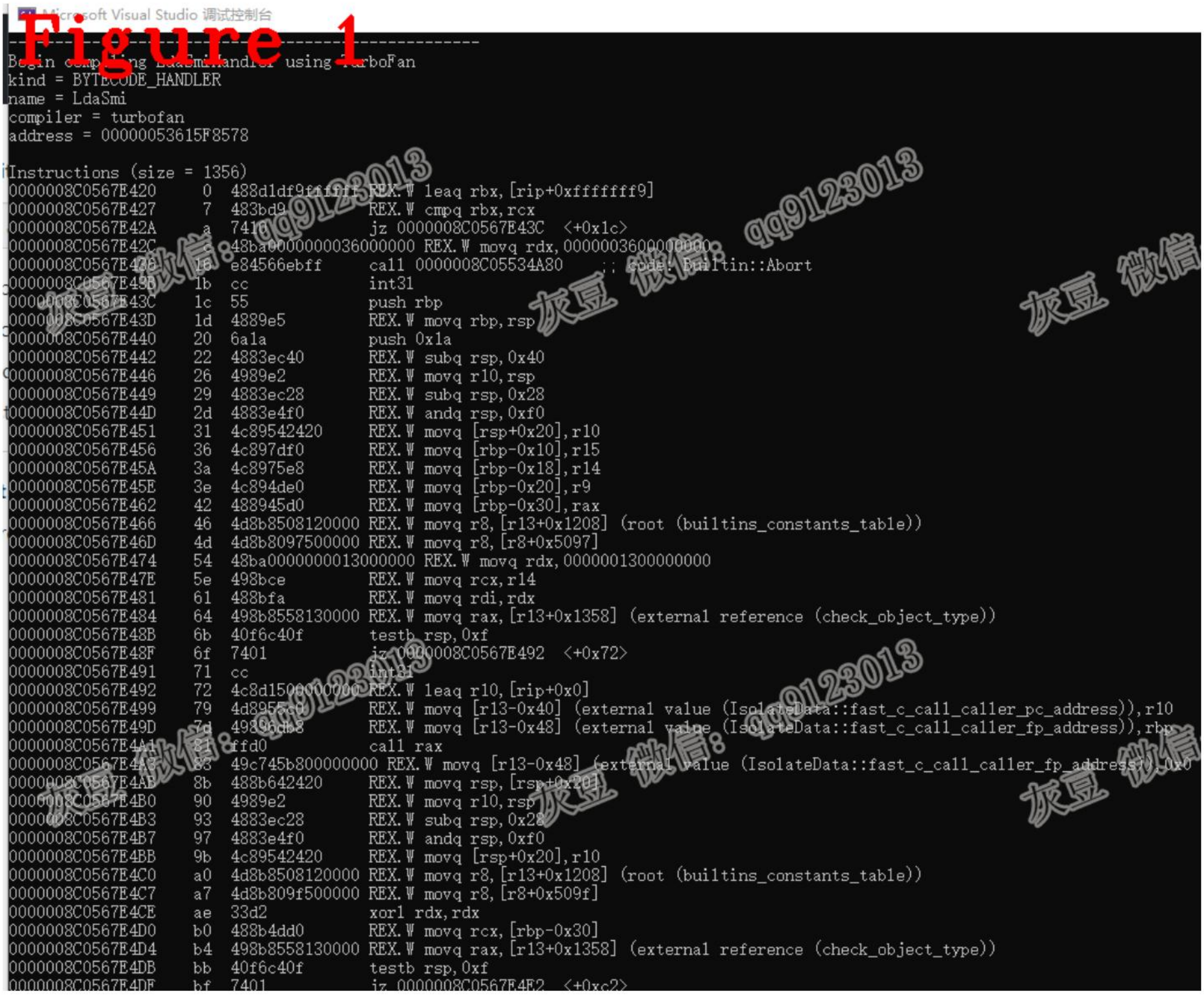
```
class V8_EXPORT_PRIVATE CodeAssembler {
    void Branch(TNode<BoolT> condition,
                CodeAssemblerParameterizedLabel<T...>* if_true,
                CodeAssemblerParameterizedLabel<T...>* if_false, Args... args) {
      if_true->AddInputs(args...);
      if_false->AddInputs(args...);
      Branch(condition, if_true->plain_label(), if_false->plain_label());
    }
    template <class... T, class... Args>
     void Goto(CodeAssemblerParameterizedLabel<T...>* label, Args... args) {
       label->AddInputs(args...);
       Goto(label->plain_label());
     }
     void Branch(TNode<BoolT> condition, const std::function<void()>& true_body,
                 const std::function<void()>& false_body);
     void Branch(TNode<BoolT> condition, Label* true_label,
                 const std::function<void()>& false_body);
     void Branch(TNode<BoolT> condition, const std::function<void()>& true_body,
                 Label* false_label);
     void Switch(Node* index, Label* default_label, const int32_t* case_values,
                 Label** case_labels, size_t case_count);
}
```

3 Compiler Pipeline

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

```
Handle<Code> CodeAssembler::GenerateCode(CodeAssemblerState* state,
                                          const AssemblerOptions& options) {
RawMachineAssembler* rasm = state->raw_assembler_.get();
Handle<Code> code;
Graph* graph = rasm->ExportForOptimization();
code = Pipeline::GenerateCodeForCodeStub(...Omit parameters...)
                .ToHandleChecked();
state->code_generated_ = true;
return code;
}
//............Separator line.......................
MaybeHandle<Code> Pipeline::GenerateCodeForCodeStub(...Omit parameters...) {
    OptimizedCompilationInfo info(CStrVector(debug_name), graph->zone(), kind);
    info.set_builtin_index(builtin_index);
    if (poisoning_level != PoisoningMitigationLevel::kDontPoison) {
      info.SetPoisoningMitigationLevel(poisoning_level);
    }
    // Construct a pipeline for scheduling and code generation.
    ZoneStats zone_stats(isolate->allocator());
    NodeOriginTable node_origins(graph);
    JumpOptimizationInfo jump_opt;
    bool should_optimize_jumps =
        isolate->serializer_enabled() && FLAG_turbo_rewrite_far_jumps;
    PipelineData data(&zone_stats, &info, isolate, isolate->allocator(), graph,
                      nullptr, source_positions, &node_origins,
                      should_optimize_jumps ? &jump_opt : nullptr, options);
    data.set_verify_graph(FLAG_verify_csa);
    std::unique_ptr<PipelineStatistics> pipeline_statistics;
    if (FLAG_turbo_stats || FLAG_turbo_stats_nvp) {
    }
    PipelineImpl pipeline(&data);
    if (info.trace_turbo_json_enabled() || info.trace_turbo_graph_enabled())
{//..Omit...
    }
    pipeline.Run<CsaEarlyOptimizationPhase>();
    pipeline.RunPrintAndVerify(CsaEarlyOptimizationPhase::phase_name(), true);
    // ............omitted............
    PipelineData second_data(...parameters omitted...);
    second_data.set_verify_graph(FLAG_verify_csa);
    PipelineImpl second_pipeline(&second_data);
    second_pipeline.SelectInstructionsAndAssemble(call_descriptor);
    Handle<Code> code;
    if (jump_opt.is_optimizable()) {
      jump_opt.set_optimizing();
      code = pipeline.GenerateCode(call_descriptor).ToHandleChecked();
    } else {
      code = second_pipeline.FinalizeCode().ToHandleChecked();
    }
    return code;
}
```

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.



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