

《Chrome V8源码》 32.字节码和 Compiler Pipeline 的细节



1 摘要

本篇文章是 Builtin 专题的第七篇。上篇文章讲解了 `Builtin::kInterpreterEntryTrampoline` 源码，本篇文章将介绍 Builtin 的编译过程，在此过程中可以看到 Bytecode handler 生成 code 的技术细节，同时也可借助此过程了解 Compiler Pipeline 技术和重要数据结构。

2 Bytecode handler的重要数据结构

`GenerateBytecodeHandler()`负责生成Bytecode handler，源码如下：

```
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, ...) \
16.     case Bytecode::k##Name: \
17.         Name##Assembler::Generate(&state, operand_scale); \
18.         break;
19.     BYTECODE_LIST(CALL_GENERATOR);
20. #undef CALL_GENERATOR
21. }
22.     Handle<Code> code = compiler::CodeAssembler::GenerateCode(&state, options);
23. #ifdef ENABLE_DISASSEMBLER
24.     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. }

```

上述代码第 7-13 行初始化 state，state 中包括 BytecodeOffset、DispatchTable 和 Descriptor，Bytecode 编译时会使用 state。第 14-21 行代码生成 Bytecode handler 源码。第 17 行 state 作为参数传入 GenerateCode() 中，用于记录 Bytecode handler 的生成结果。下面以 LdaSmi 为例讲解 Bytecode handler 的重要数据结构：

```

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

```

上述代码将累加寄存器的值设置为 smi。展开宏 IGNITION_HANDLER 后可以看到 LdaSmiAssembler 是子类，InterpreterAssembler 是父类，说明如下：

(1) LdaSmiAssembler 中包括生成 LdaSmi 的入口方法 Genrate()，源码如下：

```

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. }

```

上述第3行代码创建 LdaSmiAssembler 实例。第4行代码把 debug 信息写入 state。

(2) InterpreterAssembler 提供解释器相关的功能，源码如下：

```

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

```

```

3. //.....省略.....
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,
21.                         compiler::TNode<IntPtrT> jump_offset);
22.    void SaveBytecodeOffset();
23.    TNode<IntPtrT> ReloadBytecodeOffset();
24.    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. };

```

上述第 5 行代码获取 BytecodeArray 的地址；第 6 行代码获取 DispatchTable 的地址；第 7 行代码获取累加寄存器的值；第 8-13 行代码用于操作寄存器；第 15-16 行代码用于调用函数前后的堆栈处理；第 17-18 行代码用于跟踪 Bytecode，其中第 18 行会调用 Runtime::Runtime_InterpreterTraceBytecodeEntry 以输出寄存器信息；第 19-20 行代码是两条跳转指令，在该指令的内部调用 Advance（第 24-26 行）来完成跳转操作；第 24-26 行代码用于获取下一条 Bytecode；第 32-42 行代码定义的成员变量在 Bytecode handler 中会被频繁使用，例如在

SetAccumulator(zero_value) 中先设置 accumulator_use_ 为写状态，再把值写入 accumulator_。

(3) CodeStubAssembler 是 InterpreterAssembler 的父类，提供 JavaScript 的特有方法，源码如下：

```

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.
21.      TNode<Smi> ToSmiIndex(TNode<Context> context, TNode<Object> input,
22.                           Label* range_error);
23.      TNode<Smi> ToSmiLength(TNode<Context> context, TNode<Object> input,
24.                             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.      {}
30.      TNode<Object> GetProperty(SloppyTNode<Context> context,
31.                               SloppyTNode<Object> receiver,
32.                               SloppyTNode<Object> name) {}
33.      TNode<Object> SetPropertyStrict(TNode<Context> context,
34.                                      TNode<Object> receiver, TNode<Object> key,
35.                                      TNode<Object> value) {}
36.      template <class... TArgs>
37.      TNode<Object> CallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
38.                               TArgs... args) {}
39.      template <class... TArgs>
40.      void TailCallBuiltin(Builtins::Name id, SloppyTNode<Object> context,
41.                           TArgs... args) { }
42.      void LoadPropertyFromFastObject(...省略参数...);
43.      void LoadPropertyFromFastObject(...省略参数...);
44.      void LoadPropertyFromNameDictionary(...省略参数...);
45.      void LoadPropertyFromGlobalDictionary(...省略参数...);
46.      void UpdateFeedback(Node* feedback, Node* feedback_vector, Node* slot_id);
47.      void ReportFeedbackUpdate(TNode<FeedbackVector> feedback_vector,
48.                               SloppyTNode<UIntPtrT> slot_id, const char*
49.                               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 利用汇编语言实现了 JavaScript 的特有方法。基类 CodeAssembler 对汇编语言进行封装，CodeStubAssembler 使用 CodeAssembler 提供的汇编功能实现了字符串转换、属性获取和分支跳转等 JavaScript 功能，这正是 CodeStubAssembler 的意义所在。

上述代码第 4-9 行实现了字符串的相关操作；第 11-18 行代码实现了类型转换；第 21-26 行实现了 ES 规范中的功能；第 27-38 行实现了获取和设置属性；第 39-43 行实现了 Builtin 和 Runtime API 的调用方法；第 45-50 行代码用于管理 Feedback；第 51-55 行实现了 IF 功能。 **(4)** CodeAssembler 封装了汇编功能，实现了 Branch、Goto 等功能，源码如下：

```

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,
21.                Label** case_labels, size_t case_count);
22. }

```

3 Compiler Pipeline

GenerateBytecodeHandler() 的第 22 行代码完成了对 Bytecode LdaSmi 的编译，源码如下：

```

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(...省略参数...)
7.      .ToHandleChecked();
8.  state->code_generated_ = true;
9.  return code;
10. }
11. //.....分隔线.....
12. MaybeHandle<Code> Pipeline::GenerateCodeForCodeStub(...省略参数...) {
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);
21.     JumpOptimizationInfo jump_opt;
22.     bool should_optimize_jumps =
23.         isolate->serializer_enabled() && FLAG_turbo_rewrite_far_jumps;
24.     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())
33.     {
34.         pipeline.Run<CsaEarlyOptimizationPhase>();
35.         pipeline.RunPrintAndVerify(CsaEarlyOptimizationPhase::phase_name(), true);
36.         // .....省略.....
37.         PipelineData second_data(...省略参数...);
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.     }

```


上述第 6 行代码进入 Pipeline 开始编译工作；第 13-29 用于设置 Pipeline 信息；第 32 行的使能标记在 flag-definitions.h 中定义，它们使用 Json 输出当前的编译信息；第 34-40 行代码实现了生成初始汇编码、对初始汇编码进行优化、使用优化后的数据再次生成最终代码等功能，**注意** 第 36 行代码省略了优化初始汇编码。图1给出了 LdaSmi 的编译结果。

```

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

[Instructions (size = 1356)]
0000008C0567E420 0 488d1df9f1f1 REX.W leaq rbx, [rip+0xffffffff9]
0000008C0567E427 7 483bd9 REX.W cmpq rbx, rcx
0000008C0567E42A a 7416 jz 0000008C0567E43C <+0x1c>
0000008C0567E42C 2 48ba9000000036000000 REX.W movq rdx, 0000003600000000
0000008C0567E42E 1e e84566ebff call 0000008C05534A80 ; Code::Builtin::Abort
0000008C0567E430 1b cc int3
0000008C0567E43C 1c 55 push rbp
0000008C0567E43D 1d 4839e5 REX.W movq rbp, rsp
0000008C0567E440 20 6a1a push 0x1a
0000008C0567E442 22 483ec40 REX.W subq rsp, 0x40
0000008C0567E446 26 4989e2 REX.W movq r10, rsp
0000008C0567E449 29 483ec28 REX.W subq rsp, 0x28
0000008C0567E44D 2d 483e4f0 REX.W andq rsp, 0xf0
0000008C0567E451 31 4c89542420 REX.W movq [rsp+0x20], r10
0000008C0567E456 36 4c897df0 REX.W movq [rbp-0x10], r15
0000008C0567E45A 3a 4c8975e8 REX.W movq [rbp-0x18], r14
0000008C0567E45E 3e 4c894de0 REX.W movq [rbp-0x20], r9
0000008C0567E462 42 483945d0 REX.W movq [rbp-0x30], rax
0000008C0567E466 46 4d3b8508120000 REX.W movq r8, [r13+0x1208] (root (builtins_constants_table))
0000008C0567E46D 4d 4d3b809f500000 REX.W movq r8, [r8+0x509f]
0000008C0567E474 54 48ba0000000013000000 REX.W movq rdx, 0000001300000000
0000008C0567E47E 5e 498bce REX.W movq rcx, r14
0000008C0567E481 61 488bfa REX.W movq rdi, rdx
0000008C0567E484 64 493b8558130000 REX.W movq rax, [r13+0x1358] (external reference (check_object_type))
0000008C0567E48B 6b 40f6c40f testb rsp, 0xf
0000008C0567E48F 6f 7401 jz 0000008C0567E492 <+0x72>
0000008C0567E491 71 cc int3
0000008C0567E492 72 4c8d1500000000 REX.W leaq r10, [rip+0x0]
0000008C0567E499 79 4d395f39 REX.W movq [r13-0x40] (external value (IsolateData::fast_c_call_caller_pc_address)), r10
0000008C0567E49D 7d 49395f38 REX.W movq [r13-0x48] (external value (IsolateData::fast_c_call_caller_fp_address)), rbp
0000008C0567E4A1 81 ffd0 call rax
0000008C0567E4A3 83 49c745b800000000 REX.W movq [r13-0x48] (external value (IsolateData::fast_c_call_caller_fp_address)), r10
0000008C0567E4AB 8b 483b642420 REX.W movq rsp, [rsp+0x20]
0000008C0567E4B0 90 4989e2 REX.W movq r10, rsp
0000008C0567E4B3 93 483ec28 REX.W subq rsp, 0x28
0000008C0567E4B7 97 483e4f0 REX.W andq rsp, 0xf0
0000008C0567E4BB 9b 4c89542420 REX.W movq [rsp+0x20], r10
0000008C0567E4C0 a0 4d3b8508120000 REX.W movq r8, [r13+0x1208] (root (builtins_constants_table))
0000008C0567E4C7 a7 4d3b809f500000 REX.W movq r8, [r8+0x509f]
0000008C0567E4CE ae 33d2 xorl rdx, rdx
0000008C0567E4D0 b0 483b4dd0 REX.W movq rcx, [rbp-0x30]
0000008C0567E4D4 b4 493b8558130000 REX.W movq rax, [r13+0x1358] (external reference (check_object_type))
0000008C0567E4DB bf 40f6c40f testb rsp, 0xf
0000008C0567E4DF bf 7401 jz 0000008C0567E4E2 <+0xc2>
  
```

技术总结

- (1) 只有 v8_use_snapshot = false 时才能在 V8 中调试 Bytecode Handler 的编译过程；
- (2) CodeAssembler 封装了汇编，CodeStubAssembler 封装了 JavaScript 特有的功能，InterpreterAssembler 封装了解释器需要的功能，在这三层封装之上是 Bytecode Handler；
- (3) V8 初始化时编译包括 Bytecode handler 在内的所有 Builtin。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 邮箱：v8blink@outlook.com

本文由灰豆原创发布

转载声明，注明出处：<https://www.anquanke.com/post/id/262468>

安全客 - 有思想的安全新媒体