《Chrome V8源码》33. Lazy Compile 的技术细节

1 摘要

本篇文章是 Builtin 专题的第八篇。本篇文章将跟踪 Bytecode 的执行过程，在该过程中讲解 Lazy Compile 的启动方式、工作流程以及重要的数据结构，同时也会介绍与 Lazy Compile 相关的 Builtin。

2 Lazy Compile 的启动

在进入 Lazy Compile 之前，先要了解 Bytecode 的执行过程，借助此过程来了解 Lazy Compile 的启动方式。源码如下：

```
1.  function ignition(s) {
2.      this.slogan=s;
3.      this.start=function(){eval('console.log(this.slogan);')}
4.  }
5.  worker = new ignition("here we go!");
6.  worker.start();
7.  //............分隔线.............
8.  --- AST ---
9.  . . FUNCTION "ignition" = function ignition
10.    . EXPRESSION STATEMENT at 106
11.    . . ASSIGN at 113
12.    . . . VAR PROXY unallocated (0000016B96A17A78) (mode = DYNAMIC_GLOBAL,
assigned = true) "worker"
13.    . . . CALL NEW at 115
```

```
14.    . . . . VAR PROXY unallocated (0000016B96A17790) (mode = VAR, assigned =
true) "ignition"
15.    . . . . LITERAL "here we go!"//..........省略.............
16.    //...............分隔线.................
17.    0000025885361EAE @    0 : 13 00             LdaConstant [0]
18.    0000025885361EB0 @    2 : c2                Star1
19.    0000025885361EB1 @    3 : 19 fe f8          Mov <closure>, r2
20.    0000025885361EB4 @    6 : 64 51 01 f9 02    CallRuntime [DeclareGlobals],
r1-r2
21.    0000025885361EB9 @   11 : 21 01 00          LdaGlobal [1], [0]
22.    0000025885361EBC @   14 : c2                Star1
23.    0000025885361EBD @   15 : 13 02             LdaConstant [2]
24.    0000025885361EBF @   17 : c1                Star2
25.    0000025885361EC0 @   18 : 0b f9             Ldar r1
26.    0000025885361EC2 @   20 : 68 f9 f8 01 02    Construct r1, r2-r2, [2]
27.    0000025885361EC7 @   25 : 23 03 04          StaGlobal [3], [4]
28.    0000025885361ECA @   28 : 21 03 06          LdaGlobal [3], [6]
29.    0000025885361ECD @   31 : c1                Star2
30.    0000025885361ECE @   32 : 2d f8 04 08       LdaNamedProperty r2, [4], [8]
31.    0000025885361ED2 @   36 : c2                Star1
32.    0000025885361ED3 @   37 : 5c f9 f8 0a       CallProperty0 r1, r2, [10]
33.    0000025885361ED7 @   41 : c3                Star0
34.    0000025885361ED8 @   42 : a8                Return
35.   //..............省略.................
36.   - length: 5
37.           0: 0x02841b4e1d31 <FixedArray[2]>
38.           1: 0x02841b4e1c09 <String[8]: #ignition>
39.           2: 0x02841b4e1c51 <String[11]: #here we go!>
40.           3: 0x02841b4e1c39 <String[6]: #worker>
41.           4: 0x02841b4e1c71 <String[5]: #start>
```

上述代码分为三部分，第一部分（1-6 行）是本文使用的测试代码，其中第 5 行会启动 Lazy Compile；第二部分（8-15 行）是测试代码的 AST；第三部分（17-41 行）是测试代码的Bytecode。我们从 Bytecode 讲起：

（1） LdaGlobal [1], [0]（21 行）使用常量池[1]中的字符串作为 Key 获取全局对象，也就是获取 ignition 函数；Star1（22 行）把 ignition 存入r1；Ldar r1（25 行）从 r1 中取出 ignition 并存入累加寄存器；

（2） LdaConstant [2]（23 行）和 Star2（24 行）把字符串“here we go!”存入 r2。

Construct r1, r2-r2, [2]（26 行）构造 ignition 函数时会启动 Compiler，源码如下：

```
1.  RUNTIME_FUNCTION(Runtime_NewObject) {
2.    HandleScope scope(isolate);
3.    DCHECK_EQ(2, args.length());
4.    CONVERT_ARG_HANDLE_CHECKED(JSFunction, target, 0);
5.    CONVERT_ARG_HANDLE_CHECKED(JSReceiver, new_target, 1);
6.    RETURN_RESULT_OR_FAILURE(
7.        isolate,
8.        JSObject::New(target, new_target, Handle<AllocationSite>::null()));
9.  }
10.  //.............分隔线..............
11.  int JSFunction::CalculateExpectedNofProperties(Isolate* isolate,
12.                                                 Handle<JSFunction> function) {
13.    int expected_nof_properties = 0;
```

```
14.    for (PrototypeIterator iter(isolate, function, kStartAtReceiver);
15.         !iter.IsAtEnd(); iter.Advance()) {
16.      Handle<JSReceiver> current =
17.          PrototypeIterator::GetCurrent<JSReceiver>(iter);
18.      if (!current->IsJSFunction()) break;
19.      Handle<JSFunction> func = Handle<JSFunction>::cast(current);
20.      // The super constructor should be compiled for the number of expected
21.      // properties to be available.
22.      Handle<SharedFunctionInfo> shared(func->shared(), isolate);
23.      IsCompiledScope is_compiled_scope(shared->is_compiled_scope(isolate));
24.      if (is_compiled_scope.is_compiled() ||
25.          Compiler::Compile(isolate, func, Compiler::CLEAR_EXCEPTION,
26.                            &is_compiled_scope)) {
27.      } else {
28.      }
29.    }
30.  }
```

上述代码分为两部分，Runtime_NewObject 中 New()（第 8 行）创建新对象，也就是创建 ignition 函数。New() 中会调用第二部分代码（11-30 行）。第 24 行代码计算 ignition 的属性时会启动 Compiler 生成并执行字节码，源码如下：

```
         00000258853621BE @    0 : 82 00 04          CreateFunctionContext [0],
[4]
         00000258853621C1 @    3 : 1a f9             PushContext r1
//...省略.............
         00000258853621E7 @   41 : a8                Return
```

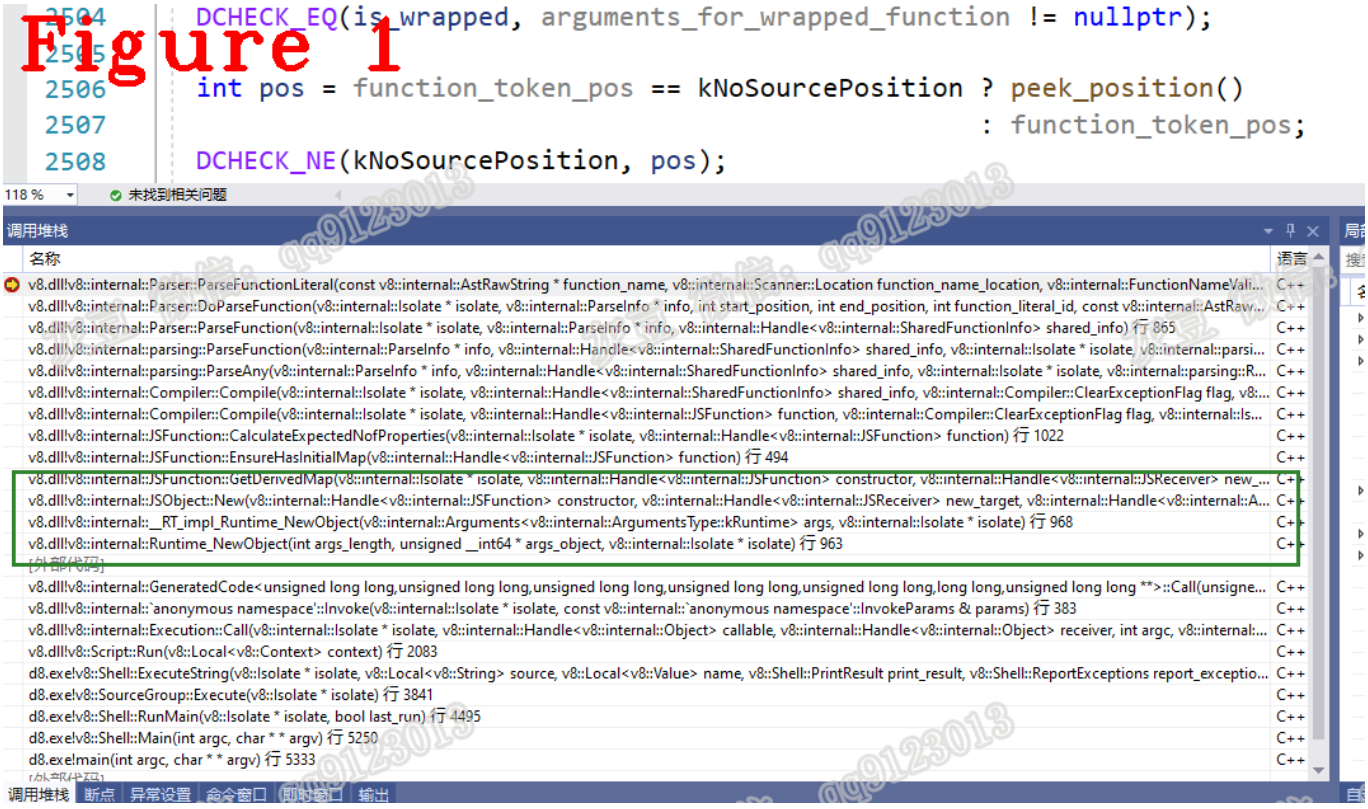
上述代码执行时不会启动 Compiler，所以 Return 指令会返回到测试代码并执行第32行 CallProperty0 r1, r2, [10]，源码如下：

```
1.  IGNITION_HANDLER(CallProperty0, InterpreterJSCallAssembler) {
2.    JSCallN(0, ConvertReceiverMode::kNotNullOrUndefined);
3.  }
4.  //.............分隔线.....................
5.    void JSCallN(int arg_count, ConvertReceiverMode receiver_mode) {
6.      Comment("sea node1");
7.      const int kFirstArgumentOperandIndex = 1;
8.      const int kReceiverOperandCount = (receiver_mode ==
ConvertReceiverMode::kNullOrUndefined) ? 0 : 1;
9.      const int kReceiverAndArgOperandCount = kReceiverOperandCount + arg_count;
10.      const int kSlotOperandIndex = kFirstArgumentOperandIndex +
kReceiverAndArgOperandCount;
11.      TNode<Object> function = LoadRegisterAtOperandIndex(0);
12.      LazyNode<Object> receiver = [=] {return receiver_mode ==
ConvertReceiverMode::kNullOrUndefined
13.                   ? UndefinedConstant() : LoadRegisterAtOperandIndex(1); };
14.      TNode<UintPtrT> slot_id = BytecodeOperandIdx(kSlotOperandIndex);
15.      TNode<HeapObject> maybe_feedback_vector = LoadFeedbackVector();
```

```
16.      TNode<Context> context = GetContext();
17.      CollectCallFeedback(function, receiver, context, maybe_feedback_vector,
18.                          slot_id);
19.      switch (kReceiverAndArgOperandCount) {
20.        case 0:
21.          CallJSAndDispatch(function, context, Int32Constant(arg_count),
22.                            receiver_mode);
23.          break;
24.        case 1:
25.          CallJSAndDispatch(
26.              function, context, Int32Constant(arg_count), receiver_mode,
27.              LoadRegisterAtOperandIndex(kFirstArgumentOperandIndex));
28.          break;//....省略.......
29.        default:
30.          UNREACHABLE();
31.      }
32.    }
33.  };
```

上述代码中，r1 寄存器的值是 JSFunction start，r2 寄存器的值是 ignition map。第 2 行代码调用 JSCallN()；第 9 行代码 kReceiverAndArgOperandCount 的值是2；第 11 行代码 function 的值是 JSFunction start；第 25 行代码 CallJSAndDIspatch() 会使用 TailCallN() 来完成函数的调用，最终进入 Lazy Compile。图 1 给出了此时的调用堆栈。

Figure 1



3 Lazy Compile

在测试代码中启动 Lazy Compile 的方式是 Runtime，源码如下：

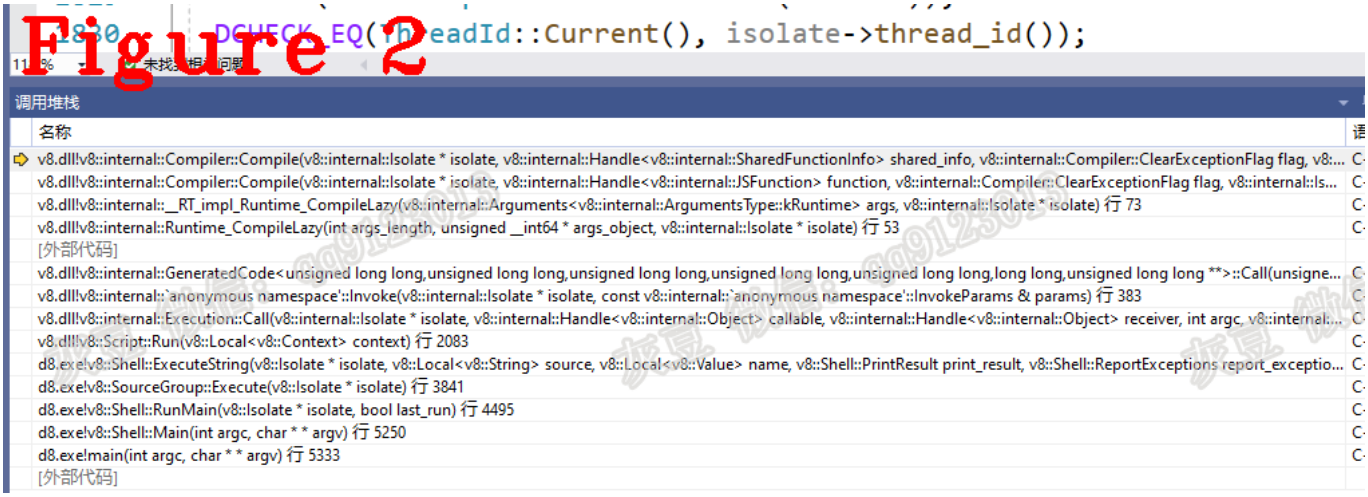
```
1.  RUNTIME_FUNCTION(Runtime_CompileLazy) {
2.    HandleScope scope(isolate);
3.    DCHECK_EQ(1, args.length());
4.    CONVERT_ARG_HANDLE_CHECKED(JSFunction, function, 0);
5.    Handle<SharedFunctionInfo> sfi(function->shared(), isolate);
6.  #ifdef DEBUG
7.    if (FLAG_trace_lazy && !sfi->is_compiled()) {
8.      PrintF("[unoptimized: ");
9.      function->PrintName();
10.      PrintF("]\n");
11.    }
12.  #endif
13.    StackLimitCheck check(isolate);
14.    if (check.JsHasOverflowed(kStackSpaceRequiredForCompilation * KB)) {
15.      return isolate->StackOverflow();
16.    }
17.    IsCompiledScope is_compiled_scope;
18.    if (!Compiler::Compile(isolate, function, Compiler::KEEP_EXCEPTION,
19.                           &is_compiled_scope)) {
20.      return ReadOnlyRoots(isolate).exception();
21.    }
22.    DCHECK(function->is_compiled());
23.    return function->code();
24.  }
```

上述代码第 3 行 function 的值是 JSFunction start；第 18 行代码启动编译流程，源码如下：

```
1.  bool Compiler::Compile(...省略....) {
2.   Handle<Script> script(Script::cast(shared_info->script()), isolate);
3.   UnoptimizedCompileFlags flags =
4.       UnoptimizedCompileFlags::ForFunctionCompile(isolate, *shared_info);
5.   UnoptimizedCompileState compile_state(isolate);
6.   ParseInfo parse_info(isolate, flags, &compile_state);
7.   LazyCompileDispatcher* dispatcher = isolate->lazy_compile_dispatcher();
8.    if (dispatcher->IsEnqueued(shared_info)) {
9.    }
10.    if (shared_info->HasUncompiledDataWithPreparseData()) {
11.    }
12.    if (!parsing::ParseAny(&parse_info, shared_info, isolate,
13.                           parsing::ReportStatisticsMode::kYes)) {
14.      return FailWithPendingException(isolate, script, &parse_info, flag);
15.    }//.........省略........
16.    FinalizeUnoptimizedCompilationDataList
17.        finalize_unoptimized_compilation_data_list;
18.    if (!IterativelyExecuteAndFinalizeUnoptimizedCompilationJobs(
19.            isolate, shared_info, script, &parse_info, isolate->allocator(),
20.            is_compiled_scope, &finalize_unoptimized_compilation_data_list,
21.            nullptr)) {
22.      return FailWithPendingException(isolate, script, &parse_info, flag);
23.    }
24.    FinalizeUnoptimizedCompilation(isolate, script, flags, &compile_state,
```

```
25.                                     finalize_unoptimized_compilation_data_list);
26.     if (FLAG_always_sparkplug) {
27.       CompileAllWithBaseline(isolate,
finalize_unoptimized_compilation_data_list);
28.     }
29.     return true;
30.   }
```

上述代码与之前讲的编译流程一致，请自行分析。**注意：** 第 27 行是 V8 新加入的编译组件，它的位置在 Ignition 和 Turbofan 之间。图 2 给出了此时的调用堆栈。



技术总结

（1） 本文涉及两次 Compile，一次用于计算对象属性，另一次是 Lazy Compile；

（2） TailCallN() 用于在当前 Block的尾部添加 Node 并完成函数调用，详见 sea of nodes。

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