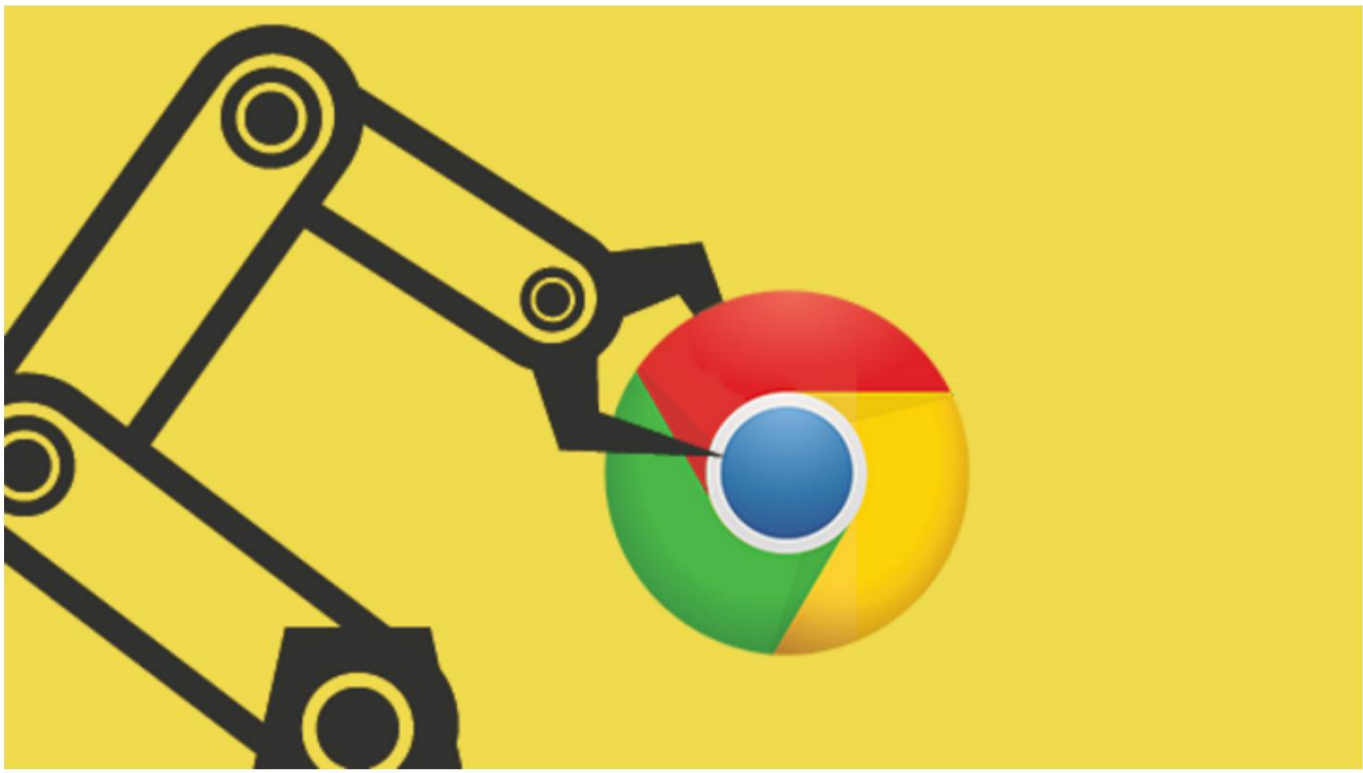
"Chrome V8 Source Code" 33. Technical details of Lazy Compile



1 abstract

This article is the eighth in the Builtin topic. This article will track the execution process of Bytecode and explain the startup of Lazy Compile in the process.

Activation methods, workflows and important data structures, and the builtin related to Lazy Compile will also be introduced.

2 Startup of Lazy Compile

Before entering Lazy Compile, you must first understand the execution process of Bytecode, and use this process to understand how Lazy Compile is started. source

The code 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();
//.....separator line.....
--- AST ---
.       .FUNCTION "ignition" = function ignition
.       . EXPRESSION STATEMENT at 106
.       .   . ASSIGN at 113
.       .   .   .   VAR PROXY unallocated (0000016B96A17A78) (mode = DYNAMIC_GLOBAL,
assigned = true) "worker"
.       .   .   .   CALL NEW at 115
```

```
14.         . . . . VAR PROXY unallocated (0000016B96A17790) (mode = VAR, assigned =
true) "ignition"
15.         . . . . LITERAL "here we go!"//.....omitted.....
16. //.....Separator line.....
17. 0000025885361EAE @ 0 : 13 00 18.                LdaConstant [0]
0000025885361EB0 @ 2 : c2 19.                       Star1
0000025885361EB1 @ 3 : 19 fe f8 20.                 Mov <closure>, r2
0000025885361EB4 @ 6 : 64 51 01 f9 02 r1-r2         CallRuntime[DeclareGlobals],

21. 0000025885361EB9 @ 11 : 21 01 00 22.            LdaGlobal [1], [0]
0000025885361EBC @ 14 : c2 23.                      Star1
0000025885361EBD @ 15 : 13 02 24.                   LdaConstant [2]
0000025885361EBF @ 17 : c1 25.                      Star2
0000025885361EC0 @ 18 : 0b f9 26.                   Ldar r1
0000025885361EC2 @ 20 : 68 f9 f8 01 02 27. 0000025885361EC7   Construct r1, r2-r2, [2]
@ 25 : 23 03 04 28. 0000025885361ECA @ 28 : 21 03   StaGlobal [3], [4]
06 29. 0000025885361ECD @ 31 : c1 30.               LdaGlobal [3], [6]
0000025885361ECE @ 32 : 2d f 8 04 08 31.            Star2
0000025885361ED2 @ 36 : c2 32. 0000025885361ED3 @   LdaNamedProperty r2, [4], [8]
37 : 5c f9 f8 0a 33. 0000025885361ED7 @ 41 :        Star1
c3 34. 0000025885361ED8 @ 42 : a8                   CallProperty0 r1, r2, [10]
35. //.....Omitted..... .                           Star0
                                                    Return

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>
```

The above code is divided into three parts. The first part (lines 1-6) is the test code used in this article, of which line 5 will start Lazy Compile; the second part The first part (lines 8-15) is the AST of the test code; the third part (lines 17-41) is the Bytecode of the test code. Let's start with Bytecode:

(1) LdaGlobal [1], [0] (line 21) uses the string in the constant pool [1] as the Key to obtain the global object, that is, obtain the ignition function number; Star1 (line 22) stores the ignition into r1; Ldar r1 (line 25) takes out the ignition from r1 and stores it in the accumulation register;

(2) LdaConstant [2] (line 23) and Star2 (line 24) store the string "here we go!" into r2.

Construct r1, r2-r2, [2] (line 26) Compiler will be started when constructing the ignition function. The source code is as follows:

```
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. //.....Separator line.....
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
twenty one.         // properties to be available.
twenty two.         Handle<SharedFunctionInfo> shared(func->shared(), isolate);
twenty three.       IsCompiledScope is_compiled_scope(shared->is_compiled_scope(isolate));
twenty four.        if (is_compiled_scope.is_compiled() ||
25.                     Compiler::Compile(isolate, func, Compiler::CLEAR_EXCEPTION,
26.                                       &is_compiled_scope)) {
27.             } else {
28.             }
29.         }
30. }
```

The above code is divided into two parts. New() (line 8) in Runtime_NewObject creates a new object, which is to create the ignition function.

The second part of the code (lines 11-30) is called in New(). When the 24th line of code calculates the properties of ignition, it will start the Compiler to generate and execute Bytecode, the source code is as follows:

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

The Compiler will not be started when the above code is executed, so the Return instruction will return to the test code and execute line 32 CallProperty0 r1, r2, [10], the source code is as follows:

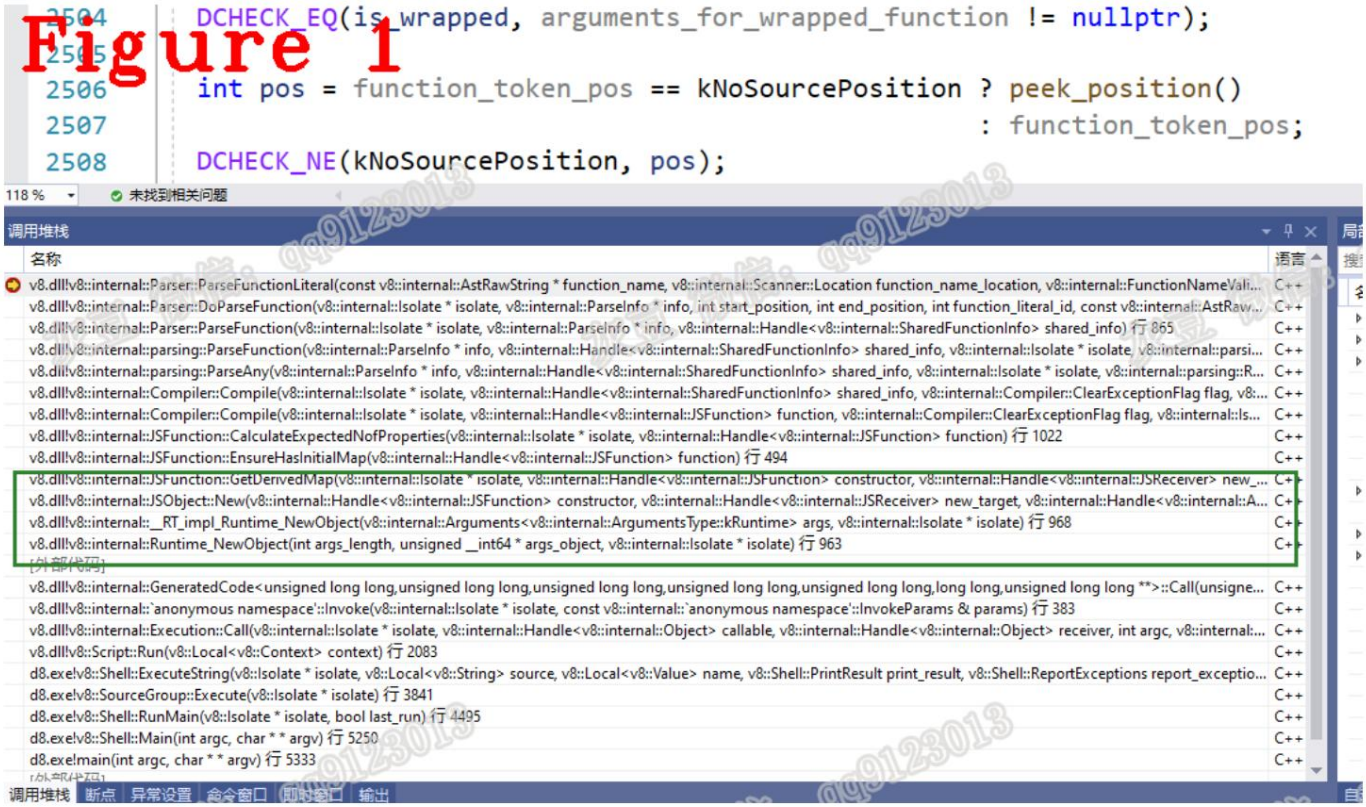
```
1. IGNITION_HANDLER(CallProperty0, InterpreterJSCallAssembler) {
2.     JSCallN(0, ConvertReceiverMode::kNotNullOrUndefined);
3. }
4. //.............Separator line.............
5.     void JSCallN(int arg_count, ConvertReceiverMode receiver_mode) {
6.         Comment("sea node1");
7.         const 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:
twenty one.             CallJSAndDispatch(function, context, Int32Constant(arg_count),
twenty two.                                       receiver_mode);
twenty three.             break;
twenty four.           case 1:
25.             CallJSAndDispatch(
26.                   function, context, Int32Constant(arg_count), receiver_mode,
27.                   LoadRegisterAtOperandIndex(kFirstArgumentOperandIndex));
28.             break;//....omitted.....
29.           default:
30.             UNREACHABLE();
31.         }
32.       }
33. };
```

In the above code, the value of the r1 register is JSFunction start, and the value of the r2 register is the ignition map. Line 2 of code calls JSCallN();

The value of kReceiverAndArgOperandCount in the 9th line of code is 2; the value of function in the 11th line of code is JSFunction start; the 25th line of code

The line of code CallJSAndDIspatch() will use TailCallN() to complete the function call and finally enter Lazy Compile. Figure 1 shows the current call stack.



3 Lazy Compile

The way to start Lazy Compile in the test code is Runtime. The source code is as follows:

```
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. }
```

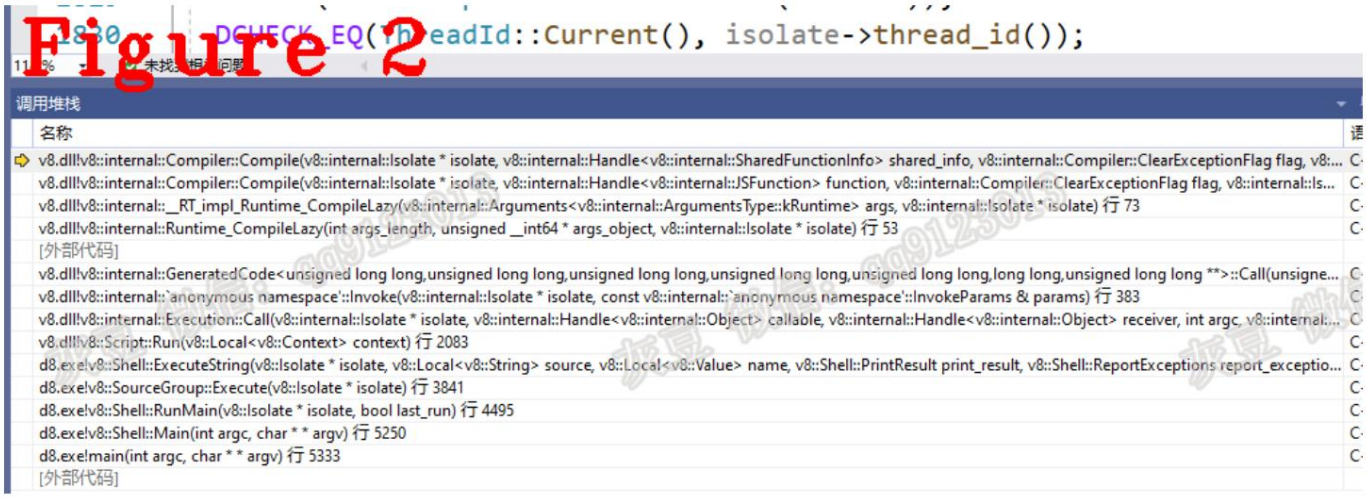
The value of function in line 3 of the above code is JSFunction start; line 18 of the code starts the compilation process. The source code is as follows:

```
1. bool Compiler::Compile(...omitted....) {
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.       }//.....omitted.....
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. }
```

The above code is consistent with the compilation process mentioned before, please analyze it yourself. Note: Line 27 is a newly added compilation component of V8. Its location is

Between Ignition and Turbofan. Figure 2 shows the call stack at this point.



Technical summary

(1) This article involves two Compiles, one for calculating object properties, and the other for Lazy Compile;

(2) TailCallN() is used to add Node at the end of the current Block and complete the function call. For details, see sea of nodes.

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

Personal abilities are limited and there are shortcomings and mistakes. Criticisms and corrections are welcome.

WeChat: qq9123013 Note: v8 Exchange Zhihu: https://www.zhihu.com/people/v8blink

This article was originally published by Gray Bean

Reprint source: https://www.anquanke.com/post/id/262578

Ankangke - Thoughtful new safety media